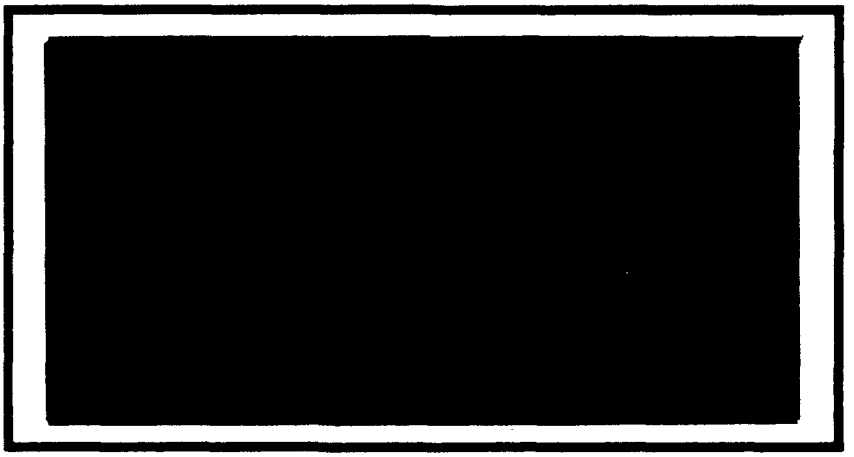

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/90D-70

AN EVALUATION OF AN Ada IMPLEMENTATION OF THE RETE ALGORITHM FOR EMBEDDED FLIGHT PROCESSORS

THESIS

F. Jesse Fanning, Captain, USAF

AFIT/GE/ENG/90D-70

Approved for public release; distribution unlimited

AFIT/GE/ENG/90D-70

AN EVALUATION OF AN Ada IMPLEMENTATION OF THE RETE ALGORITHM FOR EMBEDDED FLIGHT PROCESSORS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

F. Jesse Fanning

Captain, USAF

December 1990

Preface

This thesis grew from the Air Force's research into the use of expert systems in advanced avionics architectures. Expert systems based on Rete lie at the heart of many of the Air Force's projects in this area. This thesis addresses the suitability of an Ada implementation of Rete for an embedded avionics flight processor. Readers desiring an introduction to expert systems and Rete may refer to Appendix B. A glossary of technical terms and abbreviations used throughout this thesis is contained in Appendix A. This research was sponsored by the Avionics Laboratory, Wright Research and Development Center under In-House Project 20031112 (Fanning, 1989).

I wish to thank my thesis advisors, past and present, and the members of my committee for their continued guidance and support during the research and preparation of this thesis. Thanks also to my academic advisor, B.C. George who kept the faith throughout.

I wish to acknowledge the contributions of three individuals whose technical inputs were vital to this research: W.A. White of Barrios Technology Inc., lead programmer of CLIPS/Ada, who never tired of debugging code over the telephone and many thousands of miles; J.S. Labhart of Merit Technology Inc., who provided special insights into Rete, its implementation, and its application; and D.S. Nichols of TRW Inc., lead programmer of AARTS, who made clear the possibilities and limitations of the target environment.

Special thanks to J.L. Blair, my boss and my friend, who enabled me to advance confidently in the direction of my dreams. Thanks to J.S. Wilgus, whose care and support helped me to weather many crises. Finally, very special thanks to C.G. Coleman, whose indomitable spirit was infectious, and whose integrity as a researcher was inspiring.

F Jesse Fanning

Table of Contents

List of Figures

List of Tables

Abstract

The purpose of this research was to design and develop an expert system shell in Ada, and to evaluate the shell's execution and size performance to determine its suitability for real-time operation on the MIL-STD-1750A embedded flight processor. The expert system shell uses the CLIPS/Ada inference engine, a forward-chaining Ada implementation of Rete. The expert system shell design is presented along with an overview of the target environment-- the MIL-STD-1750A VHSIC Avionic Modular Processor (VAMP) running under the Ada Avionics Real-Time Software (AARTS) Operating System. Theoretical and empirical complexity analyses of the inference engine are presented and discussed in view of their impact on VAMP application. The performance of this inference engine was affected by five parameters of the knowledge base: a) the number of objects in working memory, b) the structural complexity of the objects and rules, c) the number of rules which share object match patterns, d) the number of match patterns per rule, and e) the number of objects bound to a match pattern. The inference engine's execution response time was found suitable for real-time operation on the VAMP; however, its memory requirement was not.

AN EVALUATION OF AN Ada IMPLEMENTATION OF THE Rete ALGORITHM FOR EMBEDDED FLIGHT PROCESSORS

I. Introduction

The Air Force has a need for expert systems which can operate in real time on embedded flight computers. Mil-STD-1750A, the Air Force's standard flight computer architecture, however, places severe constraints on applications software processing. Software programs must fit within the 1750A's 256 Kwords of core memory and execute quickly enough to solve life-critical problems, characteristics uncommon to expert systems. The definition of real time used in this research is performing a task (i.e. solving a problem) in a specified amount of time.

The Wright Research and Development Center's Avionics Laboratory identified a need for an expert system which could execute on the Air Force's next-generation implementation of the MIL-STD-1750A flight processor architecture, the VAMP (VHSIC Avionic Modular Processor, or V1750A). This expert system also must be written in Ada and be able to run under the V1750A's operating system AARTS (Ada Avionics Real-Time Software). Rete, an optimized forward-chaining inference system developed by Dr Charles Forgy of Carnegie Mellon University, shows promise as a candidate for the embedded application. Rete draws its efficiency from two basic assumptions about expert systems: many conditions are shared among production rules (structural similarity); and only a small

percentage of changes are made to working memory during any given execution cycle (temporal redundancy). The validity of these assumptions is supported by the results of this research using an Ada-based version of Rete.

Problem Statement. The Air Force has committed time, energy, and dollars to the use of Rete in time and life critical applications. Research to date, however, has been limited to specialized development systems and general purpose computers. Work must be done with Rete on the actual hardware intended for airborne applications-- the embedded flight computer. An Ada implementation of Rete must be tested on a flight computer under realistic conditions prior to serious consideration of its inclusion into the system avionics software architecture. The issues critical to future development of embedded expert systems must be identified and addressed.

Purpose. The purpose of this research was to:

- design and develop a prototype expert system shell compatible with the AARTS Operating System, and suitable for testing on an embedded flight computer, and
- evaluate the expert system shell's performance, to determine its suitability for embedded real-time flight operation on the VAMPs.

The expert system shell performance was evaluated in two areas: its ability to respond effectively under quickly changing conditions, and its ability to compile extensive rule bases into the limited amount of flight processor memory available.

Scope. This effort evaluated the performance and storage requirements of an Ada implementation of Rete for embedded flight processors and developed an embedded expert

system shell for operation on the VAMPs. The forward-chaining inference engine used was derived from NASA's CLIPS/Ada implementation of Rete. A VAMP was the target architecture for the Rete implementation, configured with 256 Kwords of RAM consisting of 16 bit words. A kernel executive resident on the host processor was provided by AARTS for scheduling of the expert system and providing it with data from the aircraft. The expert system, including working memory, discrimination net (rules), and inference engine, was to reside in the remaining flight processor memory. Due to memory model limitations of the AARTS Operating System which prevented testing of the expert system shell on the VAMP hardware, the development and testing of the expert system shell was accomplished on a VAX.

Structure of this Thesis. Chapter Two of the thesis presents background material and research in the area of expert systems and embedded computers. Chapter Three of the thesis contains the detailed approach, methodology, and design of the inference engine. Issues relating to the implementation on the embedded flight processor are discussed. Chapter Four contains the theoretical and empirical analyses of the CLIPS/Ada implementation of Rete. The results of the empirical tests supporting the theoretical analysis are presented in Chapter Five. Chapter Six contains the conclusions and recommendations for future embedded flight processor implementations. Appendix A contains a glossary of technical terms and abbreviations. Appendix B contains an overview of forward-chaining expert systems. Finally, Appendix C describes the specific changes made to NASA's CLIPS/Ada, the design decisions that were made to interface CLIPS/Ada to AARTS, and the structure of new software modules that had to be developed.

II. Background.

This chapter provides the background in expert systems research necessary for an understanding of the major issues in expert systems and real-time flight software research. The role of expert systems in today's fighter aircraft is discussed, along with current research in this area. The Rete system, the basis for this research, is presented and current Rete implementations are discussed. Readers desiring an introduction to expert systems and to how Rete differs from typical forward-chaining expert systems' inference engines may find this information in Appendix B.

Expert Systems in the Cockpit. Since the advent of the digital flight computer, combat aircraft have steadily grown from single mission fighters to flexible multi-role platforms. Reprogrammable flight software gives the aircraft increased mission capabilities. Along with this flexibility, the complexity of the cockpit environment has increased explosively. Pilots must not only be able to fly high performance aircraft, but must coordinate the information provided by on-board computers in areas such as navigation, aircraft systems management, weapons delivery, and communications. Information overload and high workloads often overtask the pilot during periods of high stress, such as aerial combat. Expert systems technology is currently being researched as a solution to this problem. Expert systems, programmed to emulate human decision making, can assume lower level tasks, allowing the pilot to "fly and fight" and make higher-level mission decisions. Expert systems are now being developed to aid the pilot in many areas, which include mission planning, as well as system, target, and threat management. The Pilot's Associate Program (Lizza, 1988), sponsored by the Wright Research and Development Center (WRDC), and the Defense Advanced Research Projects Agency (DARPA), is aimed at merging expert systems functioning in mission-critical areas into a cohesive system to aid the pilot.

Current Expert Systems Research. To date, expert systems research has been conducted in laboratory environments with specialized computers and reduced problem domains. Researchers typically use workstations and software tools specifically designed to rapidly prototype expert systems. This approach has helped bring expert systems to national attention, with fielded systems ranging from medical diagnosis to factory automation. Little attention has been given to the problems associated with running expert systems in an embedded computer environment such as that of a flight processor. There are two important aspects of flight computing: speed of execution and memory size. Execution speed in flight computing is crucial to the response time of applications software in critical situations. The memory size of flight computers is typically very limited, usually with no support for virtual memory.

The Rete Algorithm. Much emphasis has been placed on the execution efficiency of expert systems, the most important of which is the work done by Charles Forgy of Carnegie Mellon University. He has shown that a large percentage of the execution time in a typical expert system is spent matching objects in memory with their conditions for execution (Gupta and Forgy, 1983:5). A production system (a particular type of expert system) is composed of three elements: working memory, a set of if-then rules, and an iterative control loop called an inference engine. A more complete description of production systems may be found in Appendix B. In previous production systems, the Left Hand Sides (LHSs), or the "if" part of the if-then rules were matched against all objects in working memory on each loop. With the number of working memory elements in the thousands, and rules numbering in the hundreds for a real-world problem, the pattern matching was quite extensive. Dr Forgy created a scheme called Rete for improving the efficiency of the match cycle. The term Rete is derived from the greek word for net. The

efficiency of Rete is based on two principles: temporal redundancy and structural similarity (Forgy, 1979:25-26). Temporal redundancy is Forgy's observation that on any given match cycle, less than five percent of the working memory elements are modified. Structural similarity describes the commonality of matching conditions among LHSs of different rules. Rete is composed of two parts, a rule compiler and an inference engine, which take advantage of these principles.

The Rete compiler uses the common matching conditions of different rules to create a discrimination net composed of the matches required to activate the rules. The nodes of the net are themselves the test conditions for the matches. A node corresponding to a particular match may be linked to all the rule paths which contain it in their LHSs. Results of the matches are stored in alpha memory, as shown in Figure 1.

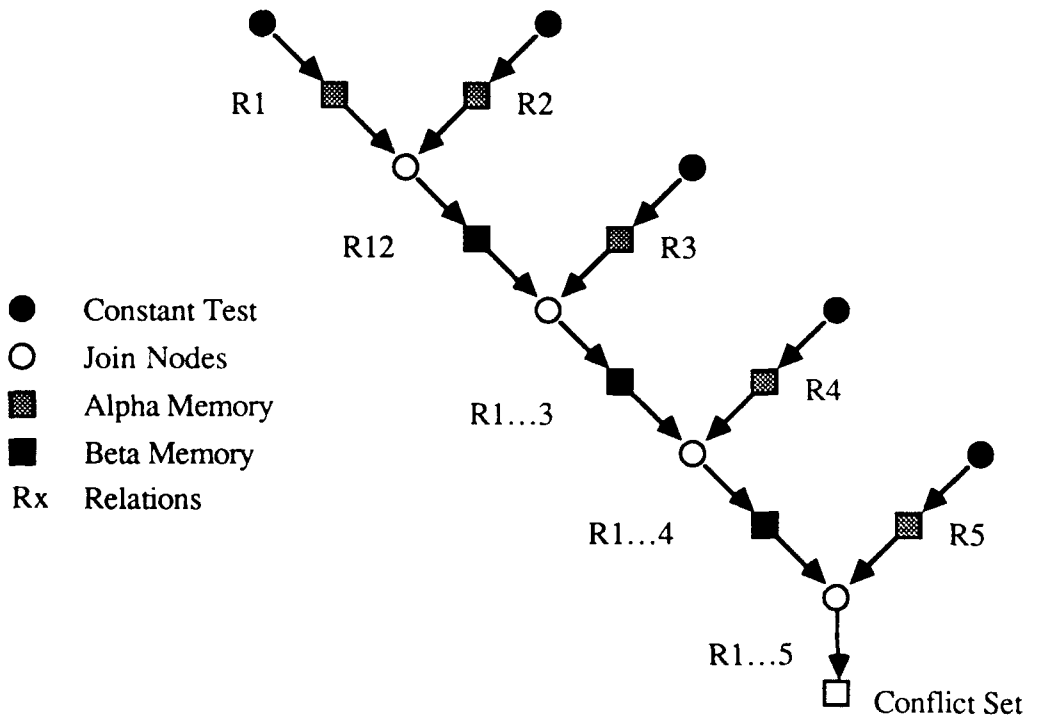

Figure 1. A Rete Network (Nayak et al., 1988:694)

The state of the alpha memory nodes reflects the partial activation of rules in the net. The LHS conditions represented by single rule-paths are brought together by join nodes. The states of the join nodes are stored in beta memory. Once all joins for a rule path are complete, a fully matched rule is placed on the conflict set. The conflict set is a list containing all rules for a given cycle that are fully matched. Selection of the rule to execute is called conflict resolution, and is performed by the inference engine.

The Rete inference engine uses temporal redundancy to increase the speed of the match cycle. Forgy found the cost of storing the match states in alpha memory to be less than that of performing the associated tests on each cycle (Forgy, 1979:48). The inference engine takes modifications to working memory, called tokens, passes them through the net, performs the match tests found at each node, and adds any resulting changes to the conflict set. After conflict resolution, the Right Hand Side (RHS), or the "then" part of the selected rule containing actions to be performed is executed, often modifying working memory. These modifications may, in turn, activate new rules, placing them on the conflict set while deactivating others, and removing them from the conflict set. Since the RHS of only one rule is executed during a loop, few working memory elements are modified. And since these modifications are the only input to the net, and they are relatively few for each cycle, the Rete inference engine attains a significant speedup in the match cycle. In fact, Rete been shown to be the most efficient approach for the many object/many pattern match problem (Gupta and Forgy, 1983; Nayak, et al., 1988).

Rete Implementation. Rete is used in a number of commercial expert system "shells" or development systems. These shells are implemented in a variety of languages, including LISP, C, and Ada, on a wide variety of hardware, including LISP machines, and mini and micro computers. NASA has developed an Ada-based version of the C Language Integrated Production System (CLIPS), an expert system development tool based on Rete.

The analysis and results presented in this thesis show that the forward-chaining inference engine of CLIPS/Ada can be embedded as a callable procedure within Ada application programs.

Several challenges must be overcome, however, before Rete can be successfully implemented on embedded flight computers due to limitations associated with flight processor architectures and a small commercial market. First, embedded flight processors such as the MIL-STD-1750A present a severe computing environment in terms of memory size, execution speed, and communications. The 1750A flight computers are restricted to 256 Kwords of RAM, with a word length of 16 bits. This amount of memory is far smaller than that available to most expert systems, which are usually implemented on computers with much greater memory available. Not only is execution speed of the expert system a concern, but this computer also has a significant communications overhead. Data from sensors and other flight computers is received over the MIL-STD-1553B communications bus. Normally encoded for transmission efficiency, the data must first be decoded for use on the host processor.

Furthermore, flight processing software must be written in Ada by mandate of the Department of Defense. Ada has some inherent limitations, such as strict data typing and lack of scheduling constructs, which make the implementation of expert systems challenging. To complicate matters, an Ada kernel executive must be present on the flight processor, reducing the 256 Kwords available for application programs. Ada's memory garbage collection strategy is implementation dependent, and may not be similar to that found in LISP, forcing explicit memory management to be handled by application programs.

Another key problem in implementing expert systems on flight processors is data consistency. Working memory is assumed by the inference engine to be static during the

selection of appropriate rules. Aircraft state data, however, is transmitted on the communications bus at relatively high rates--typically between eight and thirty-two times per second. During periods of increased activity, such as target acquisition or aerial combat, working memory objects may change at a rate that exceeds the inference engine's ability to process them. Incoming data may be filtered to reduce the impact its rate of change, but the type and extent of filtering must be determined by the application programmer to suit the problem domain.

Expert systems take considerably longer to execute (Nayak, et al., 1988), and lack the interrupt capability normally found in embedded computer software. Also, the Rete inference engine executes until the conflict set is empty, signifying problem solution. The time spent in the number of loops necessary to solve a problem may exceed the tight operational time constraint required (evading a missile, for instance), with failure yielding disastrous results.

Despite these challenges, Rete has been selected as the mechanism for expert systems on a number of Air Force Projects, including the Pilot's Associate (Lizza, 1988) and the Robotic Air Vehicle (RAV) (Lystaad, 1987). Another Air Force project, Ada Avionics Real-Time Software (AARTS), is developing an Ada executive for use on the VHSIC Avionics Modular Processor (VAMP) V1750A flight computer. An important element in the AARTS architecture is an expert system shell which is implemented in Ada and callable by the system executive. The expert system shell described in this thesis is designed to be an integral part of the AARTS system. AARTS and the VAMPs will play a key role in the configuration of future Air Force embedded flight processing.

Current State of Rete. In 1979 Forgy presented his work on Rete (Forgy, 1979), and later showed its use in an implementation for a mini-computer written in a higher order language (Forgy, 1982). Since then, Rete has been the subject of much discussion among

expert systems researchers. In 1985, the Rete-based Texas Instruments--Dallas Inference Engine (TIDIE) was developed for the Robotic Air Vehicle project by Texas Instruments, with extensions for advanced scheduling constructs (Lystaad, 1987). In 1987, Inference Corp's Automated Reasoning Tool (ART), a commercial Rete-based expert system development tool, was again chosen for prototyping the Pilot's Associate software (Lizza, 1988). Use and acceptance of Rete seemed widespread, and helped expert systems gain public attention.

Then, in 1987, Daniel Miranker published a paper challenging the efficiency of the beta-memory nodes of the Rete net, proposing an alternative approach (Miranker, 1987). His main contention was that "the size of beta memories may be combinatorially explosive" (Miranker, 1987:6). Miranker suggested removing the beta-memory nodes, and performing the matches explicitly at the joins. This procedure was proven less efficient by Nayak, et al. (1988) when compared with Rete in four representative applications.

Various implementations of Rete are currently in use, but all have the basic discrimination net and inference engine originally proposed by Forgy. Labhart, et al. (1988) have moved Rete closer to a heterogeneous processing environment by implementing the inference engine as a callable subroutine, and providing a means to update working memory externally.

Little work has been done on the structural implementation of the discrimination net, i.e. how to represent a given number of rules in the smallest amount of memory possible. This research has shown this area to be crucial to embedded flight processor applications.

Contributions. The product of this research was an Ada-based expert system shell for flight processor applications using the Rete forward-chaining inference engine in CLIPS/Ada. The design of the expert system shell is suitable for flight processor

implementation under AARTS, and serves as a vehicle for further development and research. The lessons learned during the implementation of the expert system shell under AARTS have provided feedback to the avionics community concerned with advanced software techniques and architectures for Air Force weapons systems such as the Advanced Tactical Fighter (ATF) and projects such as the Pilot's Associate. Evaluation of the NASA CLIPS/Ada inference engine was accomplished, and a list of problem areas will be provided to NASA for future enhancements of CLIPS/Ada.

Summary. While the Air Force has invested in expert systems technology to solve complex problems in the fighter aircraft arena, two important constraints of flight software must be investigated prior to successful implementation of expert systems on embedded flight computers: speed of execution, and amount of memory required. Rete, an efficient forward-chaining system, can provide an expert system suitable for such an implementation. An evaluation of a Rete-based system written in Ada able to run under the AARTS Operating System on the VAMPs was the goal of this research.

III. Design of the Expert System Shell

The approach taken in this research was to tailor the Rete-based forward-chaining CLIPS/Ada inference engine and embed it within an expert system shell suitable for operation on a VHSIC Avionics Modular Processor. The expert system shell provides the inference engine with an interface to the AARTS Operating System (AOS) services, loads the rule base, filters incoming data, and calls the inference engine to process the new data. The expert system shell and AOS interfaces were developed in VAX Ada on a Digital Equipment Corporation (DEC) VAX 11/780 and tested with its AOS shell programmer interface. This chapter presents an overview of the VAMP and AOS, and describes in detail the design of the expert system shell.

VAMP. The VHSIC Avionics Modular Processor is a VHSIC implementation of the MIL-STD-1750A Instruction Set Architecture, the standard Air Force flight processor. The VAMPs in the Integrated Test Bed (ITB) were developed by the Westinghouse Electric Corporation (WEC) under contract to the Avionics Laboratory ("Common Module Systems", undated). The VAMPs contain processor, memory, and interface modules which are plugged into and communicate over the internal PI-bus as shown in Figure 2. The PI-bus is a dual-redundant 16-bit 12.5 MHz linear bus which supports up to 32 modules.

Each CPU module contains four V1750A CPU chips, each with 256 K 16-bit words of on-board memory. Up to four CPU modules may populate a VAMP "cluster" or processor box. The V1750A has the ability to address up to one million words of extended memory over the PI-bus. There are two main Bus Interface Modules (BIMs): one to the MIL-STD-1553B bus, and the other to the fiber optic High Speed Data Bus (HSDB). The

1553B is a twisted-pair dual-redundant coaxial 1 Mbps multiplexed daisy-chain serial bus. The HSDB is a 50 Mbps dual-redundant fiber optic token-passing ring bus. The VAMP may have two separate HSDBs: the Mission Avionics Bus (MAB), for message passing between applications tasks executing on distributed VAMPs; and the Block Transfer Bus (BTB), for transmitting large blocks of data at high speeds.

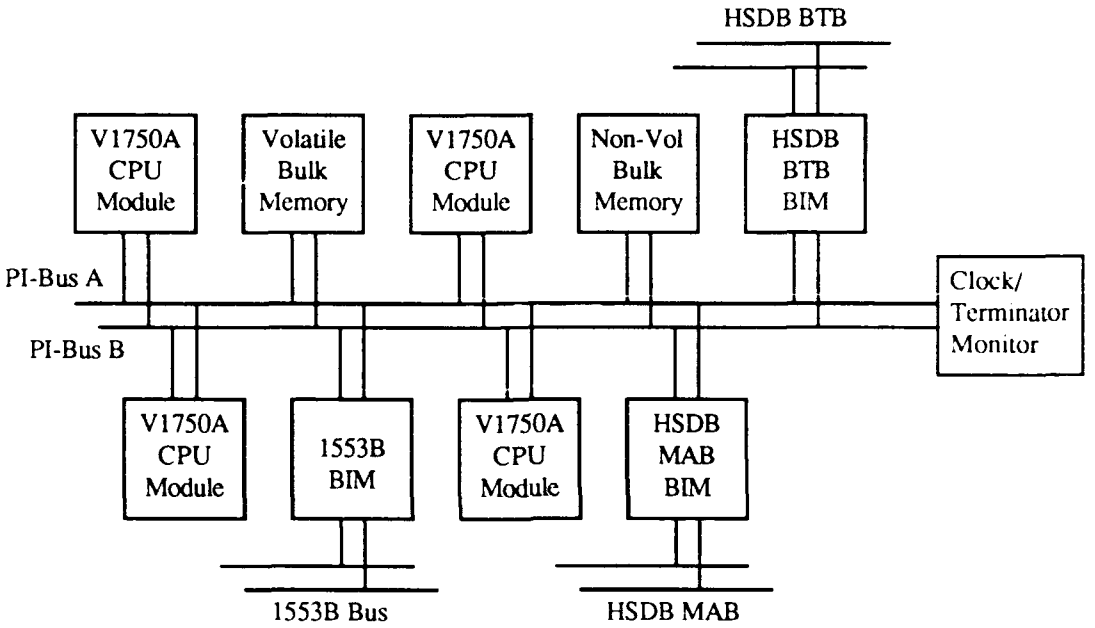

Figure 2. The VAMP Architecture

AARTS Operating System. The Ada Avionics Real-Time Software Operating System is the VAMP's kernel operating system. The AOS is being developed under contract by TRW Inc. for the Avionics Laboratory. Written in Ada, the AOS provides the system services necessary to support fault-tolerant processing on a network of distributed flight

processors. AOS services include resource management, inter-task symbolic communication and synchronization.

One CPU in a VAMP cluster is designated as the master executive, and it loads the main AOS executive. The master executive supervises execution on other CPUs in the cluster and provides graceful degradation in the event of system failure. Each of the other CPUs executes a distributed executive, a smaller version of the AOS that supervises execution of the application tasks on its CPU.

The AOS executives constantly monitor the buses and the applications tasks during operation. In the event of a CPU failure, the master executive will reconfigure the remaining CPUs to continue the mission with the available resources. Applications tasks, and even the master executive, may be reloaded onto a functional CPU and executed. This provides the system with graceful degradation.

Message passing is performed by the AOS using pre-defined message formats transmitted over the PI-bus to applications tasks on other modules. Messages to other VAMPs are passed to the MAB BIM which transmits them over the MAB HSDB. Messages to other types of avionics may be passed to the 1553B BIM which transfers them onto the 1553B bus. Messages are addressed via symbolic address tags called Message IDs (MIDs) which allow the applications to maintain communications using logical addresses while the AOS directs the messages to their physical destination.

Task synchronization is accomplished through the AOS EVENT services, which are similar to VMS event flags. An AOS event is declared and given a state (either ON or OFF). Multiple tasks may monitor and be triggered by the toggling of an event's status. Rudimentary time services are available for timed suspension of tasks as well.

The AOS is written using the TARTAN version 3.0 V1750A toolset. TRW delivered a single-CPU version to the Avionics Lab in April 1990. A multiple-CPU, single cluster

version of the AOS is expected in late December 1990, and the multiple cluster version in the summer of 1991. A VAX Ada based AOS interface package called the AOS Shell was provided by TRW for compilation and simple debugging of flight processor application modules and is hosted on a DEC VAX system.

Design of the Expert System Shell. The Expert System Shell (ES_Shell) consists of a small interface and control procedure, a rule loading and compiling module, a message receiving and object updating module, and the forward-chaining inference engine module as shown in Figure 3. The blocks represent modules of the ES_Shell, the arrows represent objects or information passed between modules, and the SMM cylinder represents System Mass Memory. In subsequent figures, double-walled blocks represent modules from CLIPS or the AOS. The ES_Shell also defines the data objects and message formats for the interface between AOS and the inference engine.

The goals in designing the ES_Shell were to develop a domain-independent shell for embedded expert system applications, to employ an inference engine written in Ada, to fit within the memory of a single V1750A CPU, and to permit expert systems to be executed in real time. The ES_Shell was designed top-down for modularity.

CLIPS/Ada. The C Language Integrated Production System (CLIPS) was created by the Software Technology Division of NASA/Johnson Space Center with support from the Air Force. CLIPS is a powerful development and delivery expert system tool which provides a complete environment for the construction of rule-based expert systems (Giarrantano, 1989). CLIPS incorporates a Rete-based forward-chaining inference engine into an expert system development tool complete with rule development and debugging capabilities. The rule syntax is based on Inference Corp's Automated Reasoning Tool (ART). When the DoD mandated the use of Ada for program development, an Ada version

was developed for NASA by Barrios Technology (Melebeck, 1989:iii). Since CLIPS/Ada is available at no cost to government agencies along with complete source code, documentation, and technical support, its inference engine was selected for this research.

The components of CLIPS/Ada used in the ES_Shell are the forward-chaining inference engine, the rule loader and Rete net compiler, and the object modification functions. The CLIPS/Ada development tool was used for rule debugging and testing.

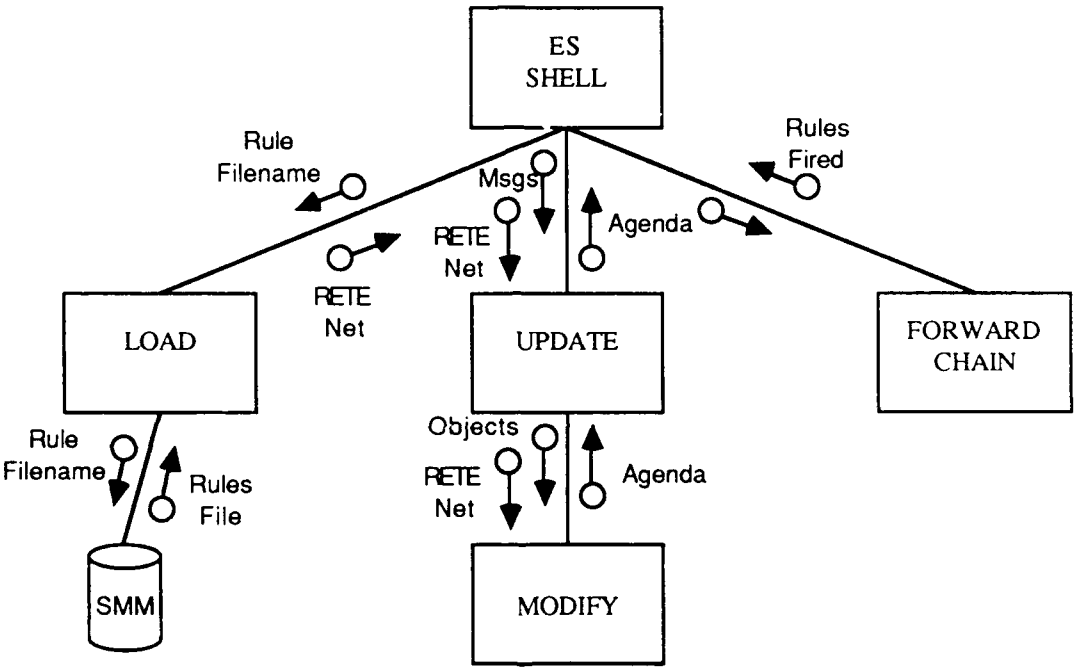

Figure 3. The ES_Shell Modules

LOAD Module. The knowledge of the expert system is encoded in a series of if-then constructs called production rules. The file containing the rules is known as the rule base. In a Rete system, the rule base is compiled into a discrimination network allowing efficient execution of the inference engine.

Using the AOS file services and the CLIPS/Ada rule compiler, the LOAD module loads the rule base and compiles it into the Rete net. The CLIPS/Ada rule compiler parses the rules and checks the syntax of the rules as it constructs the Rete net. CLIPS working memory elements are placed on the FACT_LIST. Errors in the rules are signalled and those with invalid syntax are thrown out. The LOAD module is called only once during system startup and initialization of the ES_Shell by the Operational Flight Program (OFP).

UPDATE Module. The ES_Shell was designed to operate on embedded flight processors, which operate in a real-time non-monotonic environment. Objects in the world are constantly changing, and their states must be reflected in the working memory of the expert system. The UPDATE module performs the match portion of the expert system cycle, and acts as the interface between the real world and the expert system. Object changes come to the ES_Shell in the form of AOS messages. The message formats are based on pre-defined data structures agreed upon by the ES_Shell and the sender. The messages come to the ES_Shell via the PI-bus from applications tasks on other CPUs or possibly other VAMPs through the 1553B or HSDB BIMs. The AOS posts the messages into a data buffer and signals their presence to the ES_Shell.

The UPDATE module retrieves each message from the buffer and examines it for changes. Because each change in working memory triggers activity in the Rete net, it is desirable to minimize the number of assertions to only those which convey meaningful changes. For example, repeatedly asserting the aircraft's cruise altitude of 10,000 feet is redundant since no rule would be activated. Therefore, new values are only asserted when they have changed by some domain-dependent amount. These pre-determined amounts are called filter values. Data which differs from the previously stored value by an amount greater than the filter value is asserted. The UPDATE modules are shown in Figure 4.

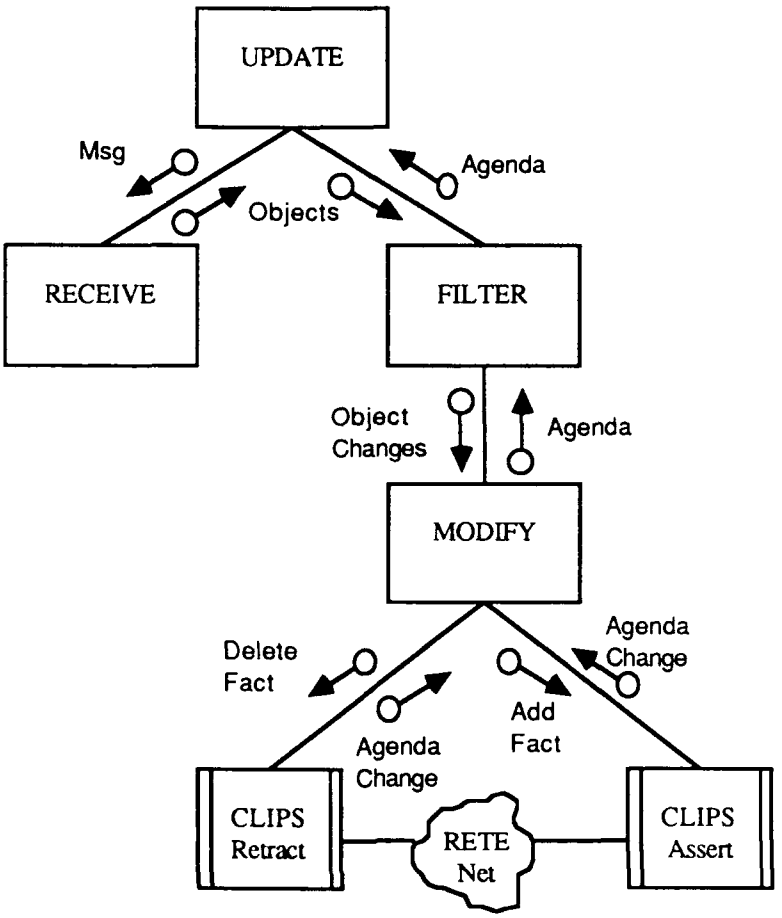

Figure 4. The UPDATE Modules

Data objects fall into two categories: statically and dynamically allocated. The majority of the aircraft attributes, such as airspeed and altitude, are known *a priori* and so are their data types (such as integer and real). Since these attributes are known, their types can be defined and space can be reserved to represent them in static data structures. This approach facilitates filtering. By defining the static data objects as an Ada enumerated type, the data structure may be traversed using the enumerated range, enhancing the domain independence of the filtering algorithm.

Dynamic data objects represent real-world entities such as targets and threats. The type of these dynamic objects cannot be pre-defined and so must be stored in a non-static

data structure. Thus, a table of lists was chosen to store dynamic data objects. Each object can be referenced through a fact pointer which is itself accessed by hashing the table. Hash collisions are stored in a linked list in that hash table position. The fact pointers refer to the inference engine's working memory object that represents the dynamic object.

Changed data objects are modified by the MODIFY module, which uses the CLIPS/Ada ASSERT and RETRACT functions. Both functions search the top level of the Rete net for a node corresponding to the object, and then the change in the object is passed through the net. Objects which are modified (e.g. a higher airspeed) are changed in working memory by first retracting the old fact, then asserting the new fact. If an object change results in the activation or deactivation of one or more rules, the conflict set, also called the agenda in CLIPS, is modified accordingly.

Once UPDATE has modified all changed objects in working memory, it returns to the ES_Shell, which calls the forward-chaining inference engine.

FORWARD-CHAIN Module. The FORWARD-CHAIN module performs the select and execute portions of the expert system cycle. The FORWARD-CHAIN module calls the CLIPS/Ada inference engine procedure as shown in Figure 5. The inference engine examines the agenda and selects an activated rule to fire.

The RHS of the rules may modify working memory as a consequence of rule firing, in turn passing data through the Rete net and possibly adding or removing other rules from the agenda via the CLIPS ASSERT and RETRACT functions. The CLIPS/Ada inference engine supports calling user-defined functions from the RHS of a rule. A critical conclusion reached in a rule firing may require immediate notification to another process. To support this need, the function SEND was developed for the expert system shell to allow the inference engine to communicate via the AOS message services. The RHS syntax for the message is (SEND MID &arguments). MID is the PI-bus message ID

number and &arguments is the argument list. The inference engine passes these arguments to the function in a linked list. The interface function extracts the data from the list, builds the outgoing message, and calls the AOS message service to transmit the message.

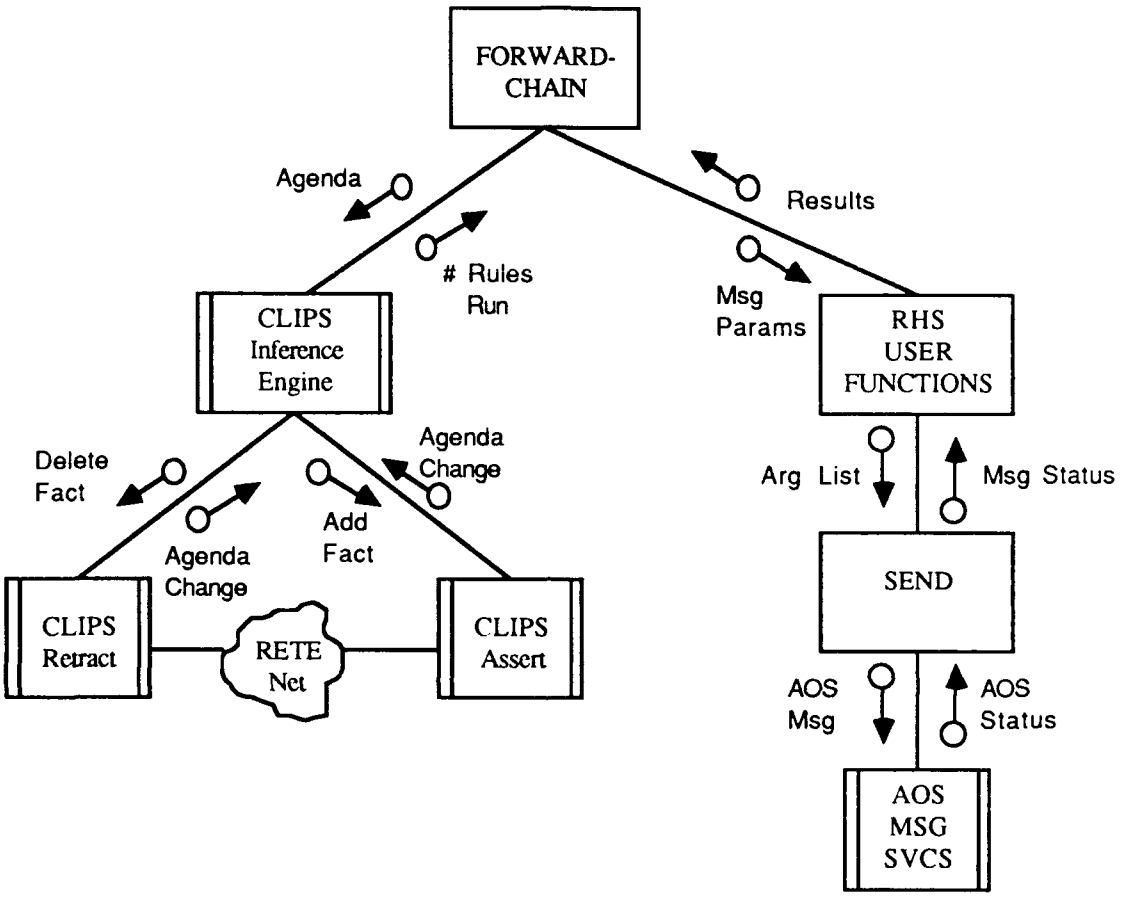

Figure 5. The FORWARD-CHAIN Modules

VAX/AOS Shell Implementation. The design for the ES_Shell was initially developed and tested on a VAX 11/780 using VAX Ada. Since both CLIPS/Ada and the AOS Shell were programmed in VAX Ada, this design was easier to debug on a VAX using the VAX Symbolic Debugger than it would have been on the VAMPS. The AOS Shell consists of a set of interface specifications and test stubs to allow applications programmers to develop and test modular code on the VAX. Not all AOS services are implemented in the AOS

Shell, but MESSAGE_IO, TIME, and EVENT services are, which allowed all of the ES_Shell/AOS interfaces except the file I/O to be tested.

The first step in the VAX implementation was to select the CLIPS/Ada functions needed to perform inferencing. Procedures were needed to load rules and compile the Rete net, to modify working memory, and to perform the forward-chaining inference. The RULES_MANAGER.LOAD_RULES procedure was used to load the rules file and compile the rules into the Rete net. The FACT_MANAGER.ASSERT and RETRACT functions were used to modify working memory. The ENGINE.RUN procedure provided the forward-chaining inference.

The second step in the VAX implementation was to select the AOS services needed to accomplish message passing, task synchronization, and process analysis. The message TRANSMIT and RECEIVE functions from the AOS MESSAGE_IO package were used for inter-task communications. The EVENT procedures were selected for task synchronization instead of the Ada task rendezvous mechanism, because EVENT services allow tasks on distributed CPUs to signal each other. The AOS TIME services were used for dynamic process analysis.

The third step in the VAX implementation was to program the ES_Shell design in VAX Ada, and to integrate it with the CLIPS/Ada and AOS Shell. A small rule base was used to verify the correct performance of the CLIPS/Ada inference engine in an embedded mode. Test driver routines were also written to examine the interaction between the ES_Shell and the AOS Shell. Because the AOS Shell contained only simple testing stubs for most services, full functional testing could not be accomplished on the VAX.

VAMP/AOS Implementation. It was not possible, without major modifications,to fit the ES_Shell executable code within the 256 Kwords memory on a V1750A CPU. The AOS

executive occupies approximately one page or 64 Kwords. In addition, four Kwords of each code segment must be allocated for AOS use, leaving just two code segments of 60 Kwords apiece with one 64 Kword data segment. The ES_Shell layout is shown in Figure 6 with respect to the other VAMP modules.

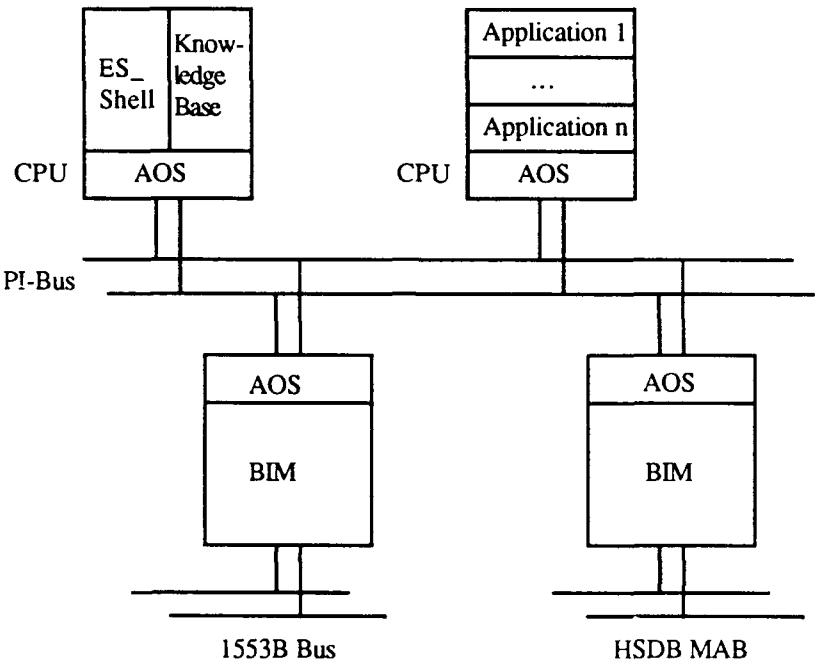

Figure 6. The VAMP/AOS Module Layout

Summary. The VAMP architecture and AOS provide an environment that encourages modular design of flight software. The ES_Shell was designed in such a fashion, drawing upon the CLIPS/Ada and AOS components necessary to produce a flexible expert system shell for flight computer applications. The initial implementation of ES_Shell on the VAX 11/780 using the Symbolic Debugger and AOS Shell facilitated testing and evaluation of the expert system shell.

IV. Analysis

This chapter contains the analysis of the expert system shell. The analysis of the expert system shell was limited to an evaluation of the CLIPS/Ada functions it used to perform the forward chaining inference. A theoretical analysis was performed on the source code, and parameters which affect the inference engine's time and space complexities were identified. These complexities were compared to Forgy's Rete analysis and tested empirically. The empirical tests were conducted using a series of benchmark tests on a MicroVAX computer. The results of these tests are presented in Chapter Five.

Analysis of the Rete Algorithm. The original analysis of the Rete algorithm was performed by Dr Charles Forgy in 1979. Worst and best case analyses were presented for both time and space complexity of Rete. The following analysis is summarized from the reprint of Dr Forgy's thesis entitled "On the Efficient Implementation of Production Systems " (Forgy, 1982).

Time Complexity of Rete. Dr Forgy identified two specific factors which adversely impact the performance of the inference engine. These are an increase in the number of productions and an increase in the number of working memory elements. Dr Forgy summarized the affect of working memory on speed in this way:

> The precise relation between system size and execution speed depends on the productions' LHSs. The LHSs with many poorly discriminating patterns, which cause many tokens to be stored, can ... cause the system to run slowly. (Forgy, 1982:45)

Space Complexity of Rete. Dr Forgy stated that there were three parameters affecting the space complexity of Rete: the number of working memory elements, the number of productions (rules), and the number of modifications to working memory elements, or tokens.

The number of working memory elements affects the size of the system in two ways. First, each working memory element requires an amount of space in the system for its storage. Second, some working memory elements may be stored in the net as tokens many times in beta memory nodes as they match distinct productions which share LHS clauses.

The number of productions has an effect on system space also. Obviously, each production requires some space in the net for the nodes to represent it. The size of the net can be reduced if rules share LHS clauses, and requires less constant test and join nodes, and their associated alpha and beta memory nodes. The RHS actions of each rule must be stored as well. As the number of rules increases, so does the size of the net.

The number of tokens, according to Dr Forgy, depends on the number of patterns in each rule and "how discriminating the patterns are" (Forgy, 1982:45). The net size will increase as the number of tokens stored in it increases.

Dr Forgy explained that ideally the size required to represent the compiled rule net should be less than or equal to the size of the uncompiled rules. In practice, however, this may be difficult to achieve. For example, the size of the compiled net in the OPS4 implementation is up to fifty percent larger than the uncompiled rules (Forgy, 1982: 45). The space and time complexities of Rete according to Dr Forgy's analysis are summarized in Table 1.

While there are two exponential terms in the analysis, these depend on every possible combination of LHS clauses to be present in the rule base. In practice, this would not usually be the case, and the number of enumerations would be much less. The performance would, in fact be closer to linear.

Table 1. Summary of Forgy's Theoretical Analysis.

	Best Case	Worst Case
Effect of Working Memory size on number of tokens	O(1)	$O(W^C)$
Effect of Production Memory size on number of nodes	O(P)	O(P)
Effect of Production Memory size on number of tokens	O(1)	O(P)
Effect of Working Memory size on time to fire one production.	O(1)	$O(W^{2C-1})$
Effect of Production Memory size on time to fire one production.	$O(\log_2 P)$	O(P)

Where C is the number of patterns in a production, P is the number of productions in production memory, and W is the number of elements in working memory (Forgy, 1982:44)

Description of the CLIPS/Ada Functions. The theoretical analysis of the CLIPS/Ada implementation is presented in two sections, an analysis of the time complexity, and an analysis of the space complexity. The theoretical analysis of the CLIPS system was performed on the CLIPS/Ada source code. Analysis tools available on the VAX were of great assistance in checking the results of this analysis.

Many sections of the CLIPS source code were examined during the development of the ES_Shell. ES_Shell used the embedded interface to the CLIPS inference engine, utilizing several CLIPS procedures for constructing, accessing, and modifying the Rete net. It became evident during program development that the two CLIPS routines which are of the most importance to the system are ASSERT and RETRACT. The inference engine control loop and routines which evaluate RHS actions are largely dominated at run time by

ASSERT and RETRACT. It is important to understand the functionality of the CLIPS ASSERT and RETRACT routines prior to presenting their complexities.

ASSERT. The function ASSERT takes a fact as input, which may represent either a new object or a new instantiation of an object. Prior to assertion into the net, the new fact is compared to each fact in the fact lists to avoid duplication. After ensuring no duplicates exist, the fact is submitted as a string and is parsed by ASSERT to extract the elements of the fact (such as the object name, its attributes, or value elements). The nature of these elements is then determined, whether they are strings themselves or numbers. String elements are hashed into a symbol table. In order to save space in the Rete net, facts stored in the net reference symbols via pointers into the symbol table. Numbers are stored directly in the fact.

Once the fact has been parsed, a data token representing the fact is driven through the net. The CLIPS/Ada system separates the constant tests from fact resolutions in two distinct nets called the pattern net and the join net. Two simple rules and their CLIPS internal representations are shown in Figure 7.

The length of a fact being asserted into the net is compared to the length of the patterns at the entry points into the pattern net. For convenience, facts of length two will be referred to as Attribute-Value (AV) pairs. Facts of length three will be referred to as Object-Attribute-Value (OAV) triples. After a token representing a fact is passed to the correct entry into the pattern net, it is passed to each of the branches stemming from that entry. If an element of the token passes the test at a node on a given branch, it will be passed on to that node's successor. Tokens which pass all the way through the pattern net will have satisfied an antecedent condition of at least one rule's LHS. These tokens are then passed to the join net.

```
(defrule engage-enemy-truck
 (?target ID truck)
 (?target IFF foe)
 (mode engage)
=>
 (launch maverick))
```

```
(defrule engage-enemy-plane
 (?target ID plane)
 (?target IFF foe)
 (mode engage)
=>
 (launch sparrow))
```

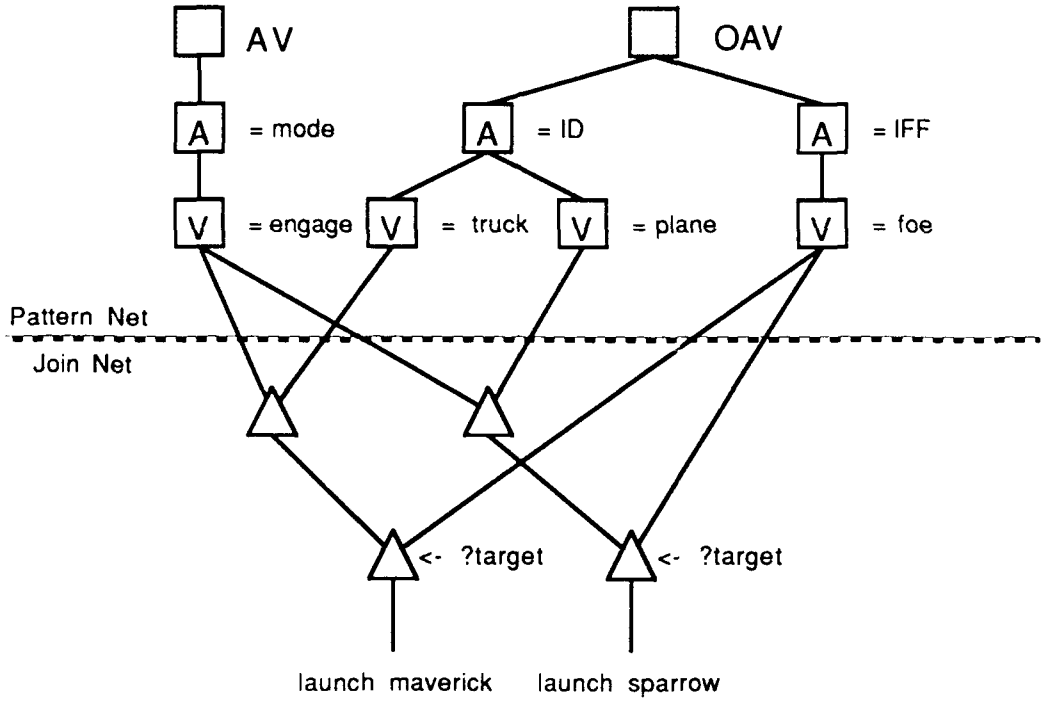

Figure 7. Sample Rules and Net

The ASSERT function drives the token into each entry in the join net pointed to by the successfully matched pattern net branch. The more rules whose LHSs share the antecedent condition, the more entries exist into the join net. In the join net, the token is bound to the rule(s) whose antecedent condition was met. Also, LHS conditions which depend on matching the same object are compared, and the bound tokens which provide a consistent match on both sides of the join are passed to the next node. For example, in the sample rule engage-enemy-truck, the pattern variable ?target will be bound to all objects with the

attributes ID and IFF. Only objects with an ID value of "truck" and IFF value of "foe" will resolve consistently, allowing the token to pass to the next join node.

When a token passes all the way through the net to the rule's final join node, the fully matched rule is said to be activated. Activated rules are placed on the conflict set, also known as the agenda. CLIPS employs a fairly simple conflict resolution strategy. A rule's firing priority is declared in the rule definition via a rule attribute called salience, expressed as a numeric value from -10,000 to +10,000. Zero is the normal or default salience if none is explicitly declared. Prior to placing an activated rule on the agenda, ASSERT searches the agenda list top-down until a rule with equal or lower salience is found. The newly activated rule is placed just before that rule. Therefore, the most recently activated rule with the highest salience is at the top of the agenda and is executed or 'fired' first.

Once a token has been passed through the net completely and any activated rules have been added to the agenda, ASSERT returns to the calling routine. Facts are stored in a list, and bindings in the join net are pointers into this list, again to save space in the Rete net.

In the event that a rule employs negative logic in its LHS, as in an antecedent condition similar to (NOT (?target ID jeep)), the ASSERT function must perform some additional tests. For each join node in the token's path, ASSERT must check all other bound tokens at that node to ensure that no objects may be consistently resolved in order to satisfy the NOT condition.

For example, let (target-1 ID ground-vehicle) and (target-1 IFF foe) exist in working memory and the NOT condition stated above be added to the sample rule engage-enemy-truck as shown in Figure 8. If the fact (target-1 ID truck) was asserted, when the NOT join node is reached, all bindings of target-1 to this node, in this case the (target-1 ID ground-vehicle) OAV triple, will be checked to ensure that (target-1 ID jeep) is not among them.

```
(defrule engage-enemy-truck
 (?target ID truck)
 (NOT (?target ID jeep))
 (?target IFF foe)
 (mode engage)
=>
 (launch maverick))
```

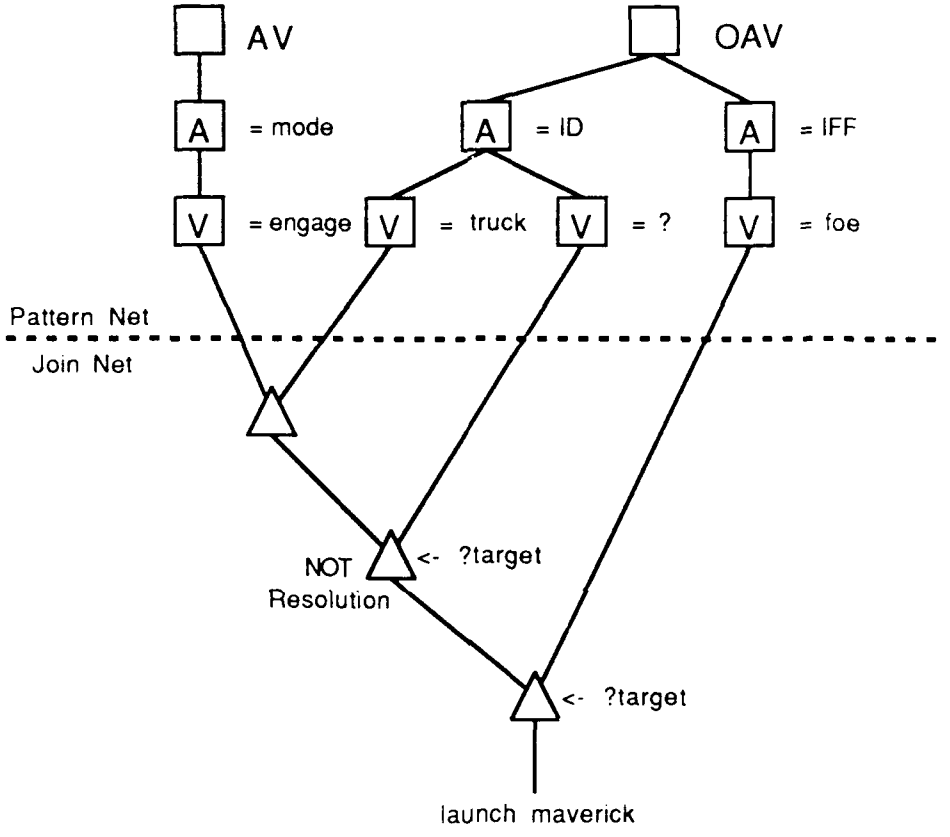

Figure 8. Sample NOT Rule and Net

RETRACT. The RETRACT function performs essentially the same steps as the ASSERT, except the token is coded as a deletion from the net instead of an addition to it. The fact list is scanned to find the fact that is to be removed. Then all of the rule bindings in the join net to which the fact is associated are removed. Finally, if a terminal join node

was bound, the rule previously activated by that node (or more specifically, the rule deactivated by the token deletion at that node) is removed from the agenda.

Both ASSERT and RETRACT functions are similar in nature and it was assumed that both would yield similar empirical results. The ASSERT function was chosen to demonstrate the remainder of the analyses.

PCA. The VAX Performance Coverage Analyzer (PCA) is a tool used on DEC VMS VAXs to examine the run-time performance of software and to assist in software test coverage. PCA will measure for a specified period the amount of CPU time spent in a particular routine. A histogram showing the time spent in each routine relative to other routines in the system may be plotted. This program was helpful in demonstrating the assumption that most of the inference engine's time was spent in ASSERT and RETRACT. During PCA analysis, RETRACT and FAST_ASSERT had the highest share of CPU times among the routines called by the inference engine.

CLIPS/Ada consists of over 700 routines in highly modular Ada packages. PCA was very helpful in narrowing the scope of the analysis by determining the specific routines which deserved the most consideration.

Source Code Analysis. The CLIPS/Ada source code was used for the actual complexity analysis of the system. A routine call chain was developed, showing all of the possible codepaths from the function FAST_ASSERT, which is the assert function called by the inference engine.

CLIPS Time Complexity. The initial attempt to determine the time complexity involved computing the time complexity of each routine called in the usual fashion (Aho, 1974). Extensive investigation of the source code revealed over 70 possible routine calls during an assert, with four singly recursive routines, and two doubly recursive routine

pairs. While an exact time complexity specification was desired for the assert function, the extremely complicated nature of the software precluded this during the period of this research, and a less rigorous approach was adopted.

Using the PCA statistics and some knowledge of the data structures and algorithms involved, five basic parameters were identified as having important impacts on the overall performance of an assert. The time complexity of the assert is proportional to:

- the number of facts in working memory,
- the relative structural complexity of the facts and the pattern net,
- the number of entries into the join net for each matched pattern,
- the number of tokens bound to the join node, and
- the length of the path from the entry into the join net to its terminal join node.

These parameters may be related to specific qualities of the knowledge base as follows:

a. Number of Facts. The number of facts will vary from expert system to expert system, and even from run to run, depending on the contents of working memory. The CLIPS/Ada assert function exhaustively searches the list of facts to ensure duplicate facts are not asserted, resulting in possible duplicate rule activations. This is a linear operation, and for novel facts contributes significantly to the response of the ASSERT.

b. Structural Complexity. The structural complexity of an object is reflected somewhat by its fact's length. Longer facts represent objects with more discriminating attributes. Because the LHS patterns of rules are designed to match with objects in working memory, the types of patterns in the antecedent clauses will resemble the objects in structure. The pattern net is partitioned by LHS pattern length. New tokens are indexed by length into these partitions when asserted. The partitioning, then, reduces the total number of entries into the pattern net to be visited, and the execution effect of this

parameter is proportional to the number of LHS patterns of the same length to be matched, which is a linear operation.

c. Number of Join Net Entries. The number of entries into the join net is a measure of how many LHS conditions share a fact pattern. During the assert, all entries into the join net must be visited, which is a linear operation.

d. Number of Fact Binds. For join nodes of LHS patterns with variable bindings or negative logic, the number of facts bound to that node is important. Each fact bound to the node must be visited to determine if its resolution with the new token would be consistent. If a resolution is consistent, the negative join fails, while the positive logic join succeeds. If there are no consistent binds, the negative join succeeds, while the positive join fails. Successful joins pass the new token to their successor nodes. Unsuccessful joins withhold the new token from further processing on that path. Because each bind must be checked at each these join nodes, the number of fact binds at each node will affect the ASSERT function in a linear fashion.

e. Length of the Join Path. The length of the join path represents the number of LHS conditions in the candidate rule. At each node in the path, a test is conducted to resolve the new token with tokens bound on the other side of the join. Obviously, the more LHS conditions in the rule, the longer the join path and therefore, the more joins which have to be performed. This parameter will affect the system's execution performance in a linear fashion.

To summarize, the execution performance of the assert depends on the amount of time spent searching the fact list for duplicate facts, plus the time spent matching against distinct LHS clauses in the pattern net, plus the time spent performing consistency tests in the join net, plus the time to activate a rule. The relationships of these parameters may be expressed as the assert's time complexity as follows:

Time Complexity of the Assert = $O(F + (S_1*C*CL) + (S_2*R*FB*RL) + (N*A))$

where F is the number of facts in the fact list, S_1 is the ratio of distinct LHS clauses to total LHS clauses, C is the number of LHS clauses, CL is the average LHS clause length, S_2 is the ratio of distinct LHS clause combinations to the total number of LHS clauses, R is the number of rules, FB is the average number of fact binds per rule that must be resolved for a successful join, RL is the average number of LHS clauses per rule, N is the number of newly activated rules, and A is the number of activated rules on the agenda.

CLIPS Space Complexity. In the space complexity analysis of CLIPS/Ada, the same approach was taken as in the source code review for the time complexity. The same parameters apply for both the positive and negative logic assert functions' space complexity.

a. Number of Facts. The facts are stored in a linked list. The linked list grows linearly with the number of facts in working memory. Each element of a fact refers to an entry in the symbol table or a numeric variable.

The symbol table is a hash table with linear linked list buckets. The table grows in size with the number of distinct symbols which hash into the same bucket. Efficient hash table storage depends on bucket conflict occurring for many fewer buckets than the total number of buckets in the table. The affect of hashing conflict should be small on the space of the overall system.

b. Structural Complexity. The structural complexity of the LHS patterns is reflected in the breadth and depth of the pattern net. Each distinct LHS clause requires an entry into the pattern. Rule bases with a variety of distinct LHS patterns will compile into a higher number of entries and therefore a broader pattern net. The length of the patterns

reflects the number of pattern tests to be performed in each distinct LHS pattern. More discriminating LHS patterns result in a deeper pattern net. Both of these relationships are linear in nature. The representation of each pattern test as a pattern node requires space in the pattern net for storage.

c. Number of Join Net Entries. The entries into the join net are stored in a linked list located at the pattern net terminal node. The entry list varies as the number of LHS clauses which share the object pattern. If it is assumed that the number of rules which share the same pattern is typically much less than the total number of rules, then in practice this effect should be small.

d. Number of Fact Binds. LHS clauses with variable bindings or negative logic require join nodes which bind the facts which have succeeded to that level in the join net. As the number of fact binds at each join node increases, so does the size required by the net. Since succeeding tokens may be passed to multiple join nodes, each requiring space for the binding, the space in the net required to represent a single fact may increase rapidly. On the average, the total number of binds will decrease toward the bottom of the join net, so this number will be much less than the number of facts times the number of join nodes.

e. Length of the Join Path. The join path length reflects the number of LHS clauses of a rule. The join path increases linearly as the number of LHS clauses per rule increases, creating a deeper join net which requires more space to represent the joins. Because typically the number of LHS clauses per rule is much less than the total number of rules, the effect of this parameter on the size of the net should be small.

To summarize, the size response of the system depends on the space required to represent the fact list, plus the space required to represent the distinct LHS pattern tests, plus the space required to test the joining of the LHS patterns, plus the space required to

represent the activated rules. The relationship of these parameters may be expressed as the assert's space complexity as follows:

$$\text{Assert Space Complexity} = O(F + (S_1*C*CL) + (S_2*R*FB*RL) + A)$$

where F is the number of facts in the fact list, S_1 is the ratio of distinct LHS clauses to the total number of LHS clauses, C is the number of LHS clauses, CL is the average number of tests per LHS clause, S_2 is the ratio of distinct LHS clause combinations to the total number of LHS clauses, R is the number of rules, FB is the number of fact bindings for the rule, RL is the average number of LHS clauses per rule, and A is the number of activated rules.

Relevance of the CLIPS Theoretical Analysis to Rete. The analysis of the CLIPS/Ada implementation indicates that in addition to the parameters of working memory and production memory size, another important parameter deserves consideration. That is the amount of sharing of LHS patterns among rules.

CLIPS Empirical Analysis. The empirical analysis of the CLIPS system was performed using a set of expert system benchmark tests on a MicroVAX computer. The results of this analysis are discussed in Chapter 5.

AIT Benchmarks. The makeup of a knowledge base is specific to the problem domain and solution method which it represents. The structural complexity of the working memory elements and rules may have a large impact on the performance of the inference engine. In order to test the relationship of the above parameters to the assert function's

performance, some generic tests were needed. The AIT expert system benchmarks address this need.

These benchmarks were developed by Artificial Intelligence Technologies, Inc. (AIT) under contract to the McDonnell Douglas Space Systems Company (MDSSC) to support MDSSC's Independent Research and Development research project entitled Real-Time Artificial Intelligence Support Systems in Ada (Knackstedt, 1990:7). The benchmarks test various parameters, including average time to assert and retract facts, and the sensitivity of the inference engine to the number of facts in working memory. The test set as a whole provides a capability to either compare the performance of different inference engines running on the same platform, or to compare the performance of various platforms running the same inference engine (Knackstedt, 1990:10).

The latter comparison is useful to future research, as the same tests may be used to compare the performance of the CLIPS/Ada inference engine on the VAMPs to that of the VAX. This comparison will be useful in predicting an expert system's performance on the VAMPs.

The AIT benchmarks generally provide a "snapshot" of the inference engine's performance under specific constraints. A series of tests was necessary to provide enough data to reasonably measure each parameter's contribution to system response. The AIT tests chosen were Test 1-2, which tests the average time per single assert and retract; Test 5-1, which measures the assert performance when LHS clauses share a pattern; and Test 8-1, which measures the assert response to ten rules with NOT conditions.

The AIT tests were modified to examine these parameters in more detail. Test ranges were extended far past the expected values for typical applications in order to examine their nature. Test 1-2 was extended to test the assert and retract functions' performances over a working memory which ranged in size from 1,000 to 10,000 instantiations of a single

object. This test was named 1-2a. The test was also extended to examine response to 1,000 to 10,000 distinct objects. The distinct objects were created by an assert rule running at a higher salience than the retract rule. This assert rule makes use of the CLIPS/Ada GENSYM function to generate new object names. GENSYM concatenates an incremented symbol count to the string "gen" (e.g. successive calls to GENSYM might produce gen1, gen2, gen3,...). This operation is of constant time complexity, and its invocation adds only a constant factor to the assert's overall time response. This second test was named 1-2b.

AIT Test 8-1 examines the assert function's performance when ten rules with NOT conditions are compiled into the net. In order to examine the function's response over a range of rule quantities, the test was modified for 1,000 to 10,000 facts in working memory with 10 binds at the negative joins. Again, a rule with higher salience was used to generate the facts. This test was named 8-1a.

AIT Test 5-1 shows the assert function's response to 200 facts which bind to a pattern in a rule's LHSs. To examine assert's response over a range of rule quantities, the test was modified to compile from 10 to 120 rules with a shared fact pattern. This test series was named 5-1a10 through 5-1a120.

The AIT tests were useful in measuring the space complexity of the CLIPS system as well. Test 5-1 was used to measure the affect of increasing working memory size and the number of shared LHS patterns. The test was performed by loading the test rules, running the test case, then measuring the amount of memory allocated to the CLIPS system. Test 5-1a10 was run with a range of 0 to 6,000 facts in working memory. The zero fact case was measured when the rules had been compiled into the net, but no facts had yet been asserted.

Tests 5-1a10 through 5-1a200 were used to show the affect of increasing the number of rules sharing LHS patterns from 10 to 200 rules. The number of facts in working memory was held constant at 400 due to memory limitations. A zero fact case was also measured to show the size of the network with no facts yet asserted.

Testing Conditions. The tests used CLIPS/Ada version 4.3 compiled with VAX Ada version 2.0. The tests were run on a DEC MicroVAX 3900 series computer, under the VAX VMS operating system version 5.3.

To facilitate measurement of the CLIPS functions on both the development and target computers, a suitable method for measuring performance on both the VAX and VAMP computers was desired. The VAX VMS operating system has many built-in methods for determining program performance (including VAX PCA), but these methods are intrinsic to VMS and would be difficult to duplicate on the VAMPs under the AOS.

AOS does provide a simple means of measuring time which makes use of the computer's system clock. The AOS TIME services include a routine which returns the value of the system clock. By inserting calls to this service at the entry and exit points of a routine, the amount of system time spent in a routine may be measured. Subtracting the entry time from the exit time yields this execution duration. Three sets of variables--time, duration, and invocation count--were created to store the results of the ASSERT, RETRACT and inference engine routine calls. The entry and exit times are stored in the time variables and their difference is stored in the duration variable. A count of the routine's invocations was incremented after the routine exit time was taken. The average time per invocation was obtained by dividing the duration by the invocation count. The smallest system clock period reported on the VAX is 10 ms, so it was necessary to invoke the routines at least 100 times to obtain a reasonable sample. This method was simple and

had the advantage of being portable to any platform which supports the AOS TIME services.

To minimize the intrusion on the CLIPS routines, a separate data reporting function was created. Only the data entry and exit time readings, and invocation count incrementing statements were actually inserted into the routines being measured. The data reporting function was designed so as to be callable from an external Ada routine, or from the RHS of a CLIPS rule.

Testing Controls. An inherent problem with this measurement method lies in the use of the system clock rather than measuring the CPU time actually spent executing the routine. The routine's duration, as measured against the system clock, may be skewed by the scheduling of tasks, time slicing, program priority, input and output, and virtual memory management.

On the VAMPs these effects are of smaller concern because the VAMPs have no virtual memory to manage, and it is simple to assure that only one task is running on the CPU and no input or output takes place. The MicroVAX, however, is a multi-user, multi-process, virtual memory computer.

To reduce the adverse impact of other processes, the MicroVAX was run with interactive logins disabled, creating a single-user computer. To minimize system process and task scheduling impacts, the test set was the only user process run, and all non-essential system processes (such as network functions) were disabled. Some overhead was incurred by the use of virtual memory, but with the physical and virtual memory limits held constant, the additional processing required for the use of virtual memory was monitored and taken into account.

To minimize the effect of memory allocation on the test measurements, knowledge of CLIPS' internal memory management was used. CLIPS requests memory in blocks from

the operating system as needed and allocates portions of this memory to internal routines on demand. CLIPS will not release memory to the operating system unless explicitly commanded. For each test series, a control run with the test case was performed in order to allocate enough memory for the test. The performance statistics were cleared and ten more runs were performed. The measurements taken on these ten runs were averaged to show the time per routine invocation over the entire test. During space complexity measurements it was noted that the size of CLIPS memory grew slightly during each run of the same test case. This growth was attributed to the accumulation of garbage memory as a result of the inference process, and was ignored because the rate of memory growth per test did not contribute greatly to the overall size of program memory.

Another problem occurred due to test case range selection. Tests with up to 10,000 facts or more than 100 rules sharing LHS patterns required amounts of memory which often exceeded the amount of physical and virtual memory available on the system. Exceeding memory limits resulted in process termination. Processes were limited to approximately 4 Mb of physical and 10 Mb of virtual memory. Some tests had to be scaled down to allow consistent data collection.

During some test cases it was difficult to vary only one parameter while holding all others constant. In these tests, the effects of other parameters were kept to a minimum, and taken into consideration in the analysis of the results. For example, Test 5-1a10-200 measures performance in response to the number of rules sharing LHS conditions. During the test series, more rules are introduced which share LHS conditions, increasing both the number of rules and the amount of sharing. To minimize the effect of more rules being introduced, the rules were made identical except for the names, requiring a minimum of pattern and join nodes, increasing only the number of terminal nodes in the join net.

The CLIPS function MEM-USED displays the number of bytes allocated by the CLIPS system. This function was used during the space complexity testing. The output of this function is not affected by other user processes or by the use of virtual memory. As was mentioned earlier, some garbage memory was accumulated as a result of the inference process, but this number was less than five percent of the total amount of memory allocated in each case, and so was ignored for test purposes.

Summary. The complexity of the CLIPS/Ada assert function is affected by five important parameters: the number of facts in working memory, the structural complexity of the facts and LHS patterns, the number of entries into the join net, the length of the join path, and for negative logic asserts, the number of facts bound at the join node. The AIT benchmark tests provided a framework for empirical analysis. These tests were extended to examine the assert's parameters in detail. The assert function's execution and size performances were bounded linearly for the tests conducted. This performance can be used as an indicator for embedded applications given the proper information about the knowledge base.

V. Results

The results of the tests supporting the empirical analysis are presented in this chapter. Each test is described and its results discussed relative to the performance of the expert system shell. The impact of the results on the embedded application is presented, along with the results of running the PES on the expert system shell.

Time Complexity. In the time complexity tests, the response of the system, in terms of execution speed, was measured against varying parameters. The parameters used were the size of working memory, the use of negative logic in the LHS, and the number of rules sharing a LHS pattern.

Number of Instantiations. In Test 1-2a, the performances of the assert, retract, and inference engine were measured against varying the number of object instantiations in working memory. An instantiation in this case was an AV pair with the same object name, but a different value. The number of instantiations of a single object ranged from 1,000 to 10,000 for the test series. The performance of the system in Test 1-2a is shown in Figure 9.

The performance of all three functions is shown in milliseconds (ms) per action versus the number of instantiations in working memory. Clearly, the performances of all three functions are linear in nature. Because the rules of this test employed a single retract for every assert, it is evident the performance of the inference engine is dominated by the retract in this test.

Number of Objects. In Test 1-2b, the performances of the assert, retract, and inference engine were measured against the number of objects in working memory. In this case, objects were AV pairs with distinct object names. The number of objects for the test

series ranged from 1,000 to 10,000. The performance of the system for Test 1-2b is shown in Figure 10.

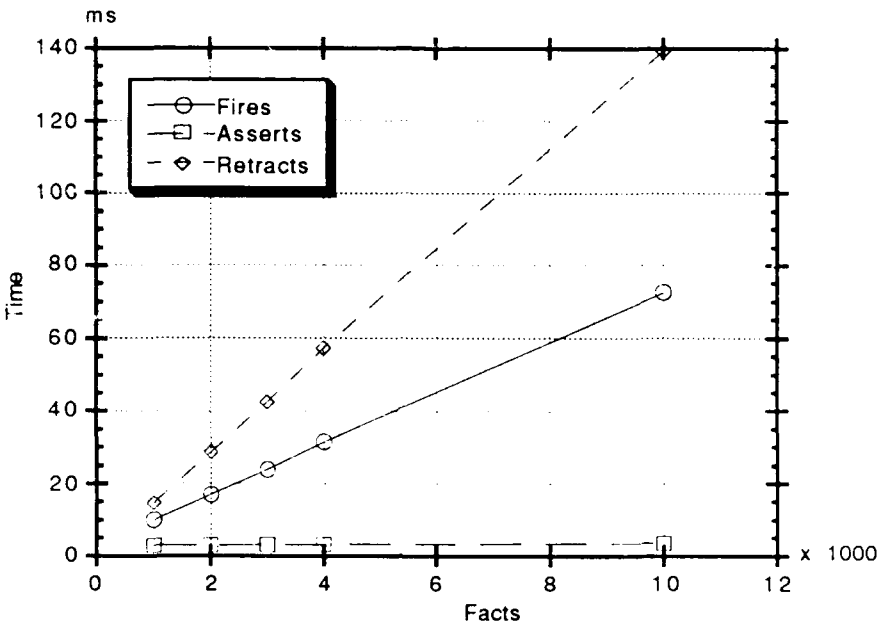

Figure 9. Test 1-2a with 1,000 - 10,000 Fact Instantiations

The performance of the system is similar to that of Test 1-2a. All three functions exhibit linear performance. As expected, the assert function's performance is offset by the use of the GENSYM function, which generated the distinct object names. The offset is evident when comparing the performances of the asserts from the two tests, shown in Figure 11. The object instantiation assert curve has a steeper slope, however, indicating additional processing taking place.

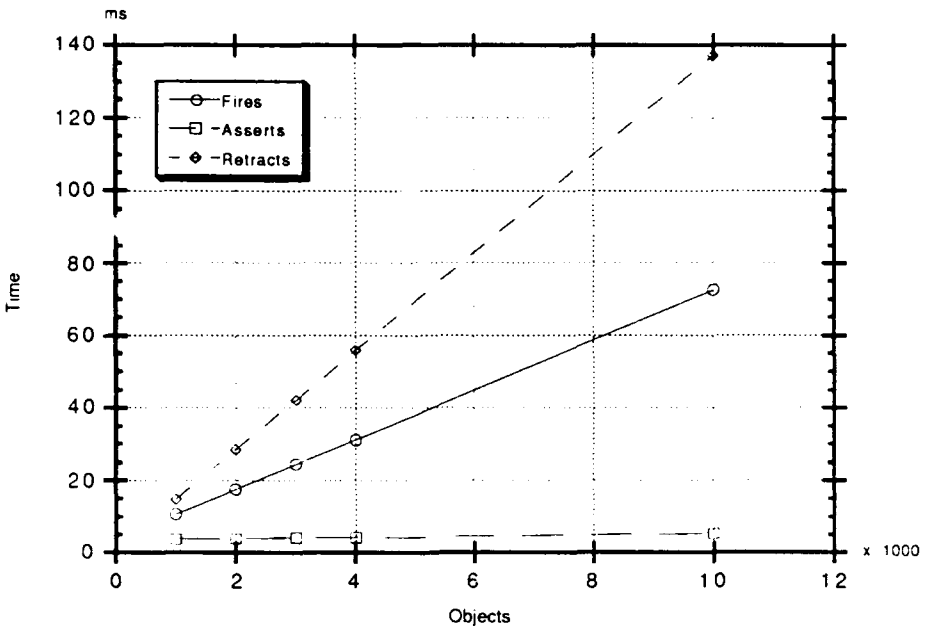

Figure 10. Test 1-2b with 1,000 - 10,000 Objects

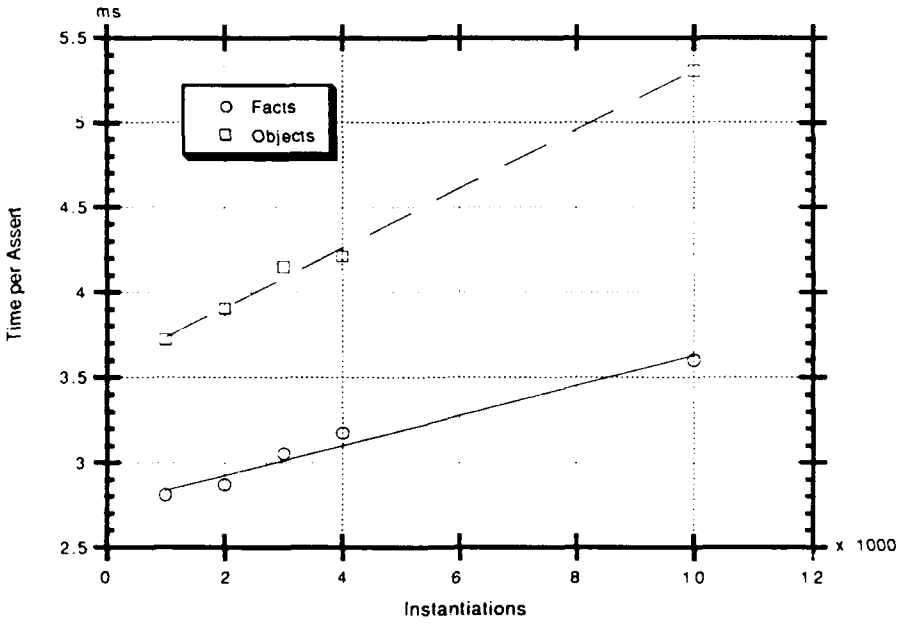

Figure 11. Assert Performance: Objects vs. Facts

Negative Logic Assert. In Test 8-1a, the performance of the assert function was measured against varying working memory size when a NOT condition was placed in a LHS pattern. Ten facts were bound to each NOT node prior to system start. The number of objects in working memory ranged from 1,000 to 10,000 objects. A comparison of the negative and positive logic asserts is found in Figure 12.

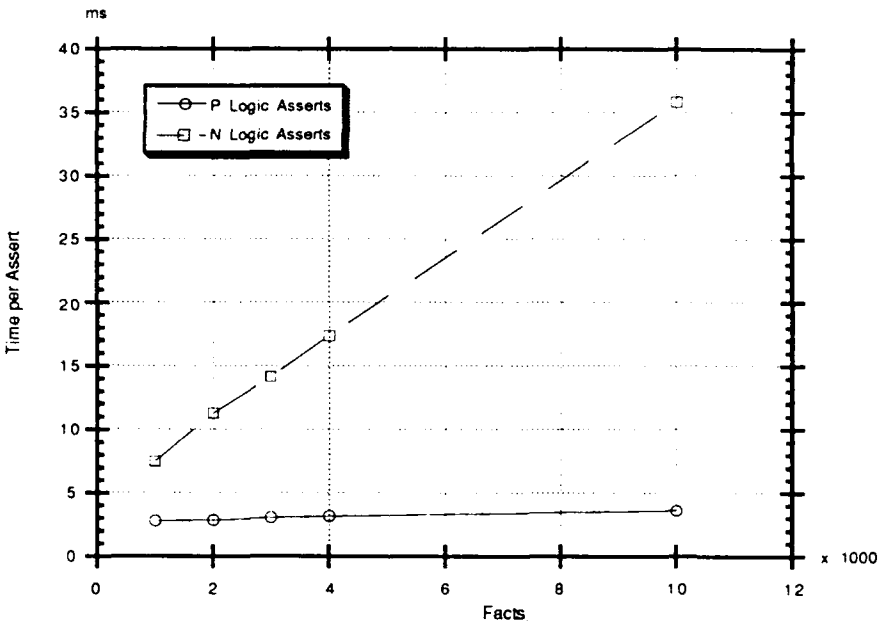

Figure 12. Positive vs. Negative Logic Asserts

The performance of the negative logic assert is linear in nature. The number of binds at each NOT node was constant for each test and so the assert function performed the same number of join bind resolution tests on each run. The steeper slope of the negative logic assert curve graphically shows that varying the number of negative join node binds

decreases the performance of the assert by requiring additional processing resulting in a larger coefficient for the linear term.

Number of Entries into Join Net. Test 5-1a measures the performance of the assert against varying the number of rules whose LHSs share an object pattern. Four hundred facts were asserted into memory for each test. The number of rules sharing LHS patterns varied from 10 to 120 rules. The performance of the assert is shown in Figure 13.

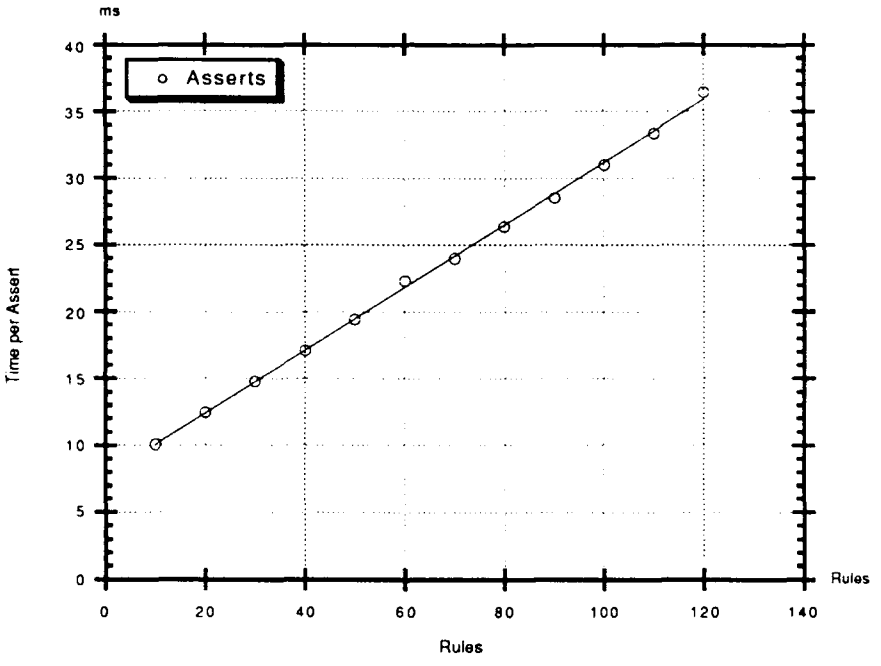

Figure 13. Test 5-1a Rules Sharing LHS Patterns

The performance of the assert is linear in nature. Since the number of facts and the number of distinct LHS clauses were held constant during the test, the linear response is attributed to the additional processing necessary to store the binds at the join nodes sharing the patterns.

Worst Case Time Complexity. The worst case time complexity of the assert routine occurred in Test 5-1a. The response in this case was influenced by the additional processing required to store the binds at the join nodes. This response corresponds to the number of rules sharing a LHS pattern. This response was still linear in nature.

Space Complexity. In the space complexity tests, the response of the system, in terms of system memory size, was measured against three parameters: the number of objects in working memory , the number of rules, and the number of rules sharing LHS patterns. All three comparisons were accomplished using the Test 5-1a with different values for the parameter of interest.

Number of Objects. The system was run with ten rules compiled into the net. The number of objects ranged from 0 to 6,000. The zero object case shows the size of the system with only the ten rules compiled which share an LHS pattern. The response of the system to this test is shown in Figure 14.

The size of the system is shown in bytes, against the number of objects in working memory. The response of the system is linear in nature. Since the slope of the line is very steep (920 bytes per fact), clearly the number of objects in working memory has an important impact on the system size.

Number of Rules. The performance of the system, in terms of size, was measured against the number of rules compiled into the net. The zero rule case shows the system's base size of 3150 bytes. The zero object case represents the size of the system with the rules compiled into the net. The number of rules ranged from 10 to 100. Each rule contained two LHS clauses and one RHS clause of length five. All rules shared one LHS pattern. The response of the system to this test is shown in Figure 15.

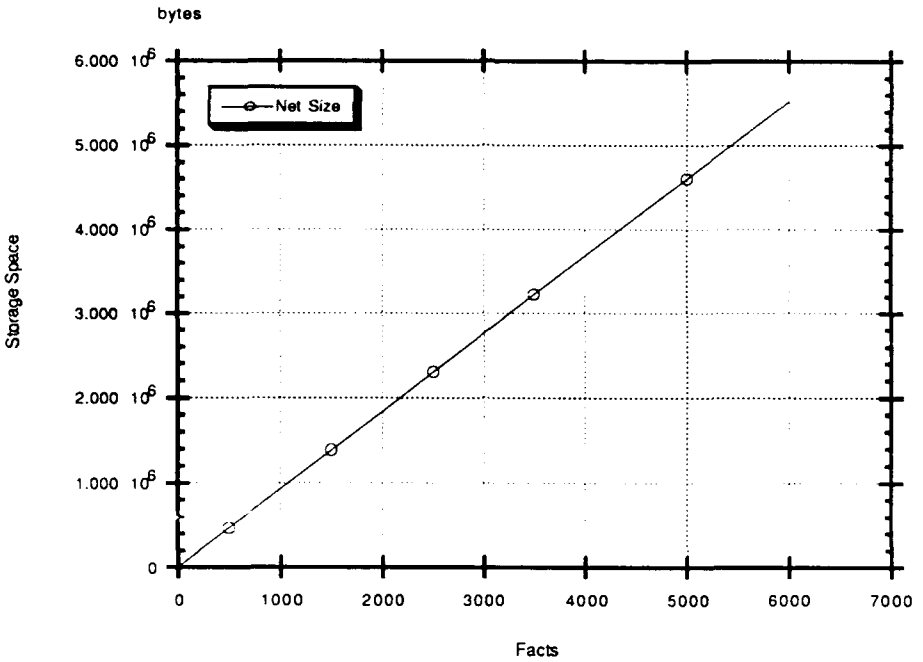

Figure 14. System Space with 10 Rules Compiled

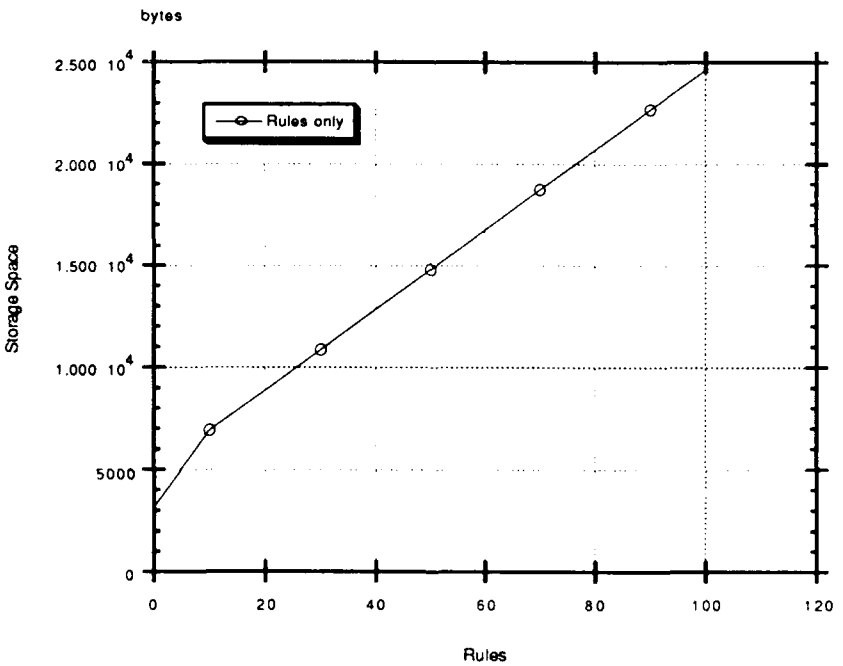

Figure 15. System Space with 0 Facts and 10 - 100 Rules

The size of the system is shown in bytes, measured against the number of rules compiled into the net. The response of the system to increasing the number of rules is linear in nature. The slope of the line is steep (197 bytes per rule); clearly the number of rules also has an important impact on the system size.

Number of Join Net Entries. The performance of the system, in terms of size, was measured against the number of entries into the join net. The number of entries into the join net ranged as the number of rules sharing a LHS pattern--from 10 to 200 rules. Each rule contained two LHS clauses and one RHS clause of length five. All rules shared one LHS pattern. Only 400 facts were asserted for each case due to memory limitations. The response of the system to this test is shown in Figure 16.

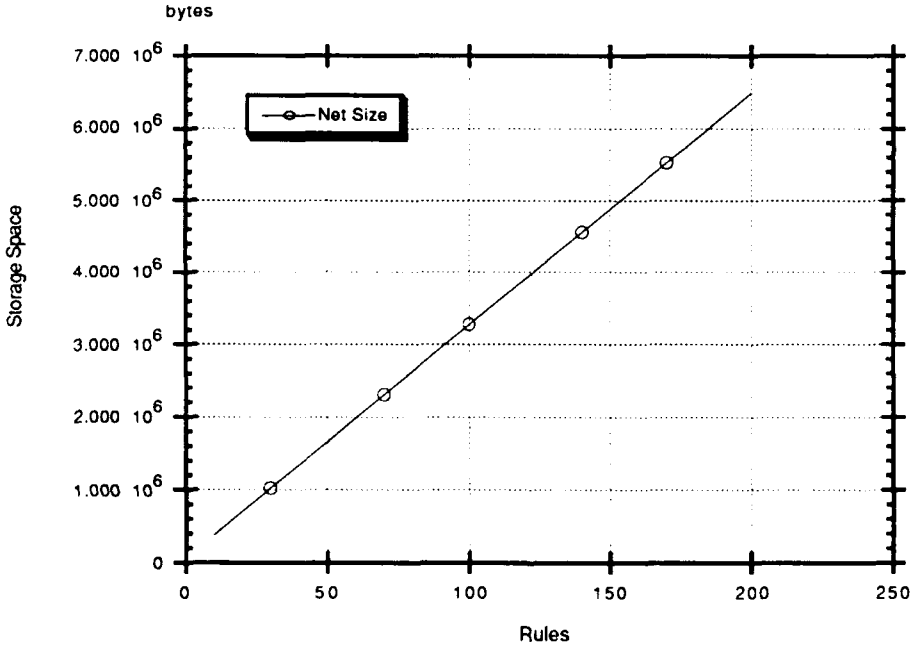

Figure 16. System Space with 400 facts and 10 - 200 Rules Sharing LHS Patterns

The response of the system is shown in bytes measured against the number of rules sharing a LHS pattern. The response of the system is linear in nature to varying this parameter. The curve is steep (32,197 bytes per rule, each storing 400 tokens). The number of rules sharing a LHS pattern is clearly important to the size response of the system.

Worst Case Space Complexity. The worst case size response of the system to these tests was linear in nature. While the nature of this response may seem encouraging, the slope of the response indicates that knowledge bases may easily exceed the finite memory of a flight computer.

Impact on System Response. Out of the parameters selected for test, the number of entries into the join net showed the highest impact overall. Both the execution and size responses involving this parameter were linear in nature, but had larger slopes than the other parameters tested. It is important to note, however, that even an average size knowledge base requires huge amounts of memory to compile and run.

Knowing the complexities of a particular inference engine implementation, and the parameters for a particular knowledge base, a good prediction of average system performance may be made. This prediction would require both an exact complexity specification of the inference engine and a thorough analysis of the knowledge base.

Implications for Embedded Processing. For operation on the VAMPs, the results are mixed. The execution response of the functions are bounded linearly. The response of the assert is within that of a typical avionics control application (Blair, 1990). The prediction of the system's execution all the way through an inference chain, however, would require

more testing with the specific knowledge base in order to guarantee a solution response time for a particular problem.

Unfortunately, the size response of the functions indicates a problem for processing on the VAMP. The slope of the space response curves indicate that CLIPS is unsuitable for the execution of even small knowledge bases since the VAMPs have only 64 Kwords of RAM. This response indicates that larger, more complicated problems requiring similarly complicated inferencing could not be run on the VAMPs. These knowledge bases would simply not have enough room to run. It may be possible to partition the knowledge base contextually and use only that portion required at a given point during execution to reduce the size of the net . With an exact specification of the space complexity of the inference engine, and a detailed analysis of the knowledge base, recommendations could made be as to the size of the knowledge base that would fit within memory, and the average number of asserts or bindings possible before exceeding memory limitations.

An Average Case. For the RAV PES, a subset of rules representing instrument takeoff procedures was rehosted in CLIPS format and run. The relatively small subset consists of 20 rules--less than twelve percent of the total rules in the PES rule base (Shakely, 1987:61). There were sixty-five objects in the knowledge base. This system required 12,594 bytes to represent the objects, 35,189 bytes to represent the rules alone, and 7,480 bytes for the fact bindings at system initialization. The total size of 55,263 bytes takes over one-third of the available data space on the VAMPs.

The size of the compiled net representing the rules themselves is 5.28 times larger than the size of the uncompiled rules. This is much larger than the ratio of 1.5 measured by Forgy for OPS4. If this ratio holds over the entire rule base it would take 448,758 bytes to

represent all of the PES rules. This is over three times more space than one data segment on the VAMP contains.

Since fact bindings are pointers stored in a link list at the join nodes, we may assume it takes at least eight bytes to represent a fact bind in the net on the VAX. This means the 20 rules in the TAKEOFFS rule base contain 935 bindings to the 65 facts at initialization or 46.75 fact bindings per rule. If this ratio holds for the entire rule base there would be 8,275 bindings requiring 66,198 bytes at system initialization. This number can only be expected to increase during system execution.

The execution response of the TAKEOFFS knowledge base running under the ES_Shell on the MicroVAX was 41.5 ms per rule fire after seventy rules fired. The rule fires consisted of 196 asserts averaging 18.7 ms per assert, and 75 retracts averaging 3.29 ms per retract. The TAKEOFFS knowledge base has a variety of fact and LHS pattern lengths, and number of LHS clauses per rule. If this knowledge base is considered typical for an avionics control application, its response can be considered an average case for CLIPS/Ada when compared to the benchmarks test results.

Summary. The results indicate that the CLIPS/Ada inference engine may be fast enough to run a typical application on the VAMP embedded flight processor. If the complexity of the knowledge base in use is known, a prediction of the real-time performance can be made. The space required by CLIPS for the discrimination net, however, is too large to fit on a VAMP flight processor, even for an average size application.

VI. Conclusions and Recommendations

The main conclusion that can be reached from the results of the theoretical and empirical analyses of CLIPS/Ada is that this implementation of Rete can be run in real time. The definition of real time in this research is to accomplish a task (i.e. solving a problem) in a specified amount of time. With the complexity parameters presented in Chapter Four, a knowledge base may be analyzed and solution times predicted. A major problem in this regard, however, is that hard requirements for solution times in many airborne problems are not known, or not specified in the general literature. The worst case time complexity of the ASSERT function of the CLIPS implementation was linear for the parameters tested. But, if this complexity and some information about the specific knowledge base to be used are known, the performance of the ASSERT may be predicted. With some additional research the performance of the overall system may be predicted as well.

The space complexity of the CLIPS/Ada implementation of Rete does, however, preclude it from serious consideration for operation on the VAMPs. While the worst case space complexity is linear in nature for the tests conducted, the slope of the response is so steep that even a small knowledge base requires more memory than is available on current flight processors.

An exact theoretical analysis of CLIPS/Ada was difficult due to the complicated nature of the software. The ASSERT function alone has seventy possible routines in its call chain, including four singly recursive routines, and two double recursive routine pairs.

The parameters which affect the time and space complexities of CLIPS/Ada ASSERT function were found. Empirical analysis was performed using these parameters that demonstrated their relationship to system response.

The internal memory management of CLIPS/Ada mechanism required fewer operating system calls for memory allocation. This internal management should increase overall performance of the system.

During the development of ES_Shell, the embedded expert system shell, it became evident that for an embedded non-monotonic environment, the inference engine must allow external routines to access and modify working memory elements. The CLIPS/Ada implementation's shortcomings in this area forced an arbitrary partitioning of working memory into elements maintained internally and elements maintained externally. Internal manipulation of designated externally maintained objects is allowed by the inference engine, but if this takes place, unexpected results may occur. Under version 4.3 of CLIPS/Ada, the designation of object maintenance is left to the application programmer who must rely on self-discipline to enforce the partitioning. Programmer enforcement is clearly undesirable in an embedded environment, so a better means of controlling access to CLIPS facts must be found.

The AOS treatment of data caused some problems in the ES_Shell development as well. Due to data security considerations, the AOS does not allow sharing of data structures among tasks in separate logical address spaces. To increase system performance, allocating the update and inferencing tasks to separate CPUs is desirable. This allocation is not possible, because both tasks would require direct access to the Rete net. Access to the net would have to be performed via AOS MESSAGE services. The additional overhead of implementing access with message passing could potentially decrease system performance.

CLIPS/Ada uses Ada exceptions for error handling and reporting. With more than 700 routines in the system, some economizing took place where a single exception handler may service ten to fifteen routines. Exceptions propagated up several levels may lose

meaning at the level handled. For example, during an assert, trying to de-reference a null access object raised a CONSTRAINT_ERROR exception at the level where it occurred. The exception handler was several levels up, however, and it reported "assert is not a valid function." This error message is not only misleading, but incorrect. This kind of exception reporting slowed the development effort.

The state of AOS software and VAMP hardware development made transition of the ES_Shell to the target environment impossible during the period of research. The VAMPs are Advanced Development Machines (ADMs), and were often at the manufacturer for repair and upgrade. The memory model limitations of the version of AOS available during this research, and slips in delivery of the next version were also major roadblocks in the transition.

The performance of the development model of the ES_Shell, however, demonstrates that an expert system running in Ada can be embedded in a large complex system involving heterogeneous problem solving techniques. The ES_Shell complements the OFP design, and can be used as an integral part of the avionic architecture in the ITB.

The use of a development methodology was helpful in moving the expert system from design to delivery. The methodology uses van de Goor's hierarchy of design levels for hardware--architecture, implementation, and realization (van de Goor, 1989:4-7). The VAMPs are a good example of this hierarchy in practice for hardware design. As shown in Figure 17, the design and development of an expert system for real-time embedded application may follow the same hierarchy.

In expert systems development, architecture design usually takes place during rapid prototyping. The output of the rapid prototype is a system behavior specification, that is the knowledge base and its interfaces to other entities in the system. During implementation design, the expert system is developed further as well as interfaces to the

system hardware and software. The output of this phase is a specification for the real-time behavior of the system. In the realization phase, the software is developed which will run on the actual real-time embedded system. This is similar to the WEC ADM realization of the VAMP implementation in hardware. The use of a software development model such as this is essential in the development of an expert system which will be run in a real-time embedded environment.

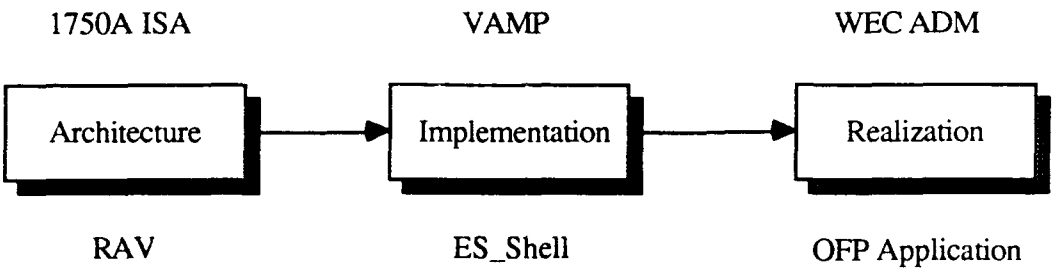

Figure 17. Real-time Expert System Development Model

A potential problem uncovered in the implementation of the ES_Shell is that CLIPS/Ada has no provision for modifying a fact in place in working memory. A modification of a fact involves first retracting the old fact, then asserting the new fact. This procedure destroys and recreates data structures which are reusable. In addition, the retract and assert operations may make changes to the agenda. The retract and assert operations may potentially deactivate, then reactivate the same rule in turn. This would degrade the overall system performance in a non-monotonic environment.

Overall, it appears that Rete is a good choice for the PES application. The execution performance of the PES TAKEOFFS knowledge base falls well within the bounds of CLIPS/Ada's worst case performance. If the PES is considered a typical avionics control application, then Rete may be useful in other control applications as well.

Recommendations. Using the parameters for system performance specified in Chapter Four, a capability for analyzing knowledge base complexity yielding predictions for real-time performance should be developed. Further theoretical analysis of the specific Rete implementation is necessary, however for an accurate performance prediction.

Because of CLIPS/Ada's space complexity, MIL-STD-1750A embedded flight processor application is unlikely. Development of a context-switchable Rete net for embedded use could alleviate space problems on the target computer. Further research must be conducted into data consistency and context resolution for this approach to be successful. Development of such a capability has potential for execution performance payoffs as well, since the Rete subnet in context would likely have less entries into its join net.

Forgy stated that for the ideal implementation of Rete the amount of space necessary to store the compiled rule net would be less than or equal to the size of the uncompiled rule base (Forgy, 1982:45). Implementation of Forgy's network representation that approaches this ratio needs to be accomplished. It is unclear whether CLIPS/Ada could be modified to take advantage of this representation.

A production-quality inference engine should be developed with the size and speed requirements of embedded flight processing in mind. The inference engine should at least use a knowledge representation consistent with the other levels of design. A consistent rule syntax would allow the knowledge base to flow between the three design levels. A consistent syntax would also facilitate maintenance and debugging of the knowledge base on the prototyping or development platforms with their suite of development and analysis tools.

Until a production-quality inference engine is available, testing should continue with the ES_Shell utilizing CLIPS/Ada. When a version of the AOS is released with less strict memory model limitations, development of ES_Shell should continue toward VAMP integration. Small expert systems could then be tested in the ITB within a true avionics architecture.

In the development of the ES_Shell, many parts of CLIPS/Ada were examined and evaluated for possible elimination from the embedded implementation. One such part is the rule loading and Rete net compilation procedure. Rule base compilation in an embedded environment should take place prior to run time, in order to save time and space. Loading a compiled network into memory on demand should be much faster than loading the rule file, parsing the rules, and compiling the rules into the network. The space for the loader, parser, and compiler code could be saved. If the structure of the net approaches Forgy's ideal implementation, then storage of the net instead of the rule base could also yield a space savings. Use of pre-compiled nets would make the net context switching procedure more efficient as well.

Additional research should be performed to determine the feasibility of truly modifying, rather than retracting and asserting facts. This capability would have payoffs in both time and space.

Finally, a problem which made reporting of the empirical analysis of the inference engine's performance difficult was the lack of a set of standard metrics for measuring expert system performance. Specifying performance in terms of rules per second, or milliseconds per rule, is useful only when referring to a specific expert system running on a specific platform. Research should be conducted to develop a standard set of metrics to discriminate among performance reports.

Appendix A. Technical Terms and Abbreviations

1553B	MIL-STD-1553B. The standard Air Force digital serial communications bus for use with avionic flight computers.
AARTS	Ada Avionics Real-Time Software. The project to develop an Ada based kernel operating system for the VAMP flight processors (Benning, 1988).
AOS	AARTS Operating System. The collection of AARTS operating services available to a VAMP application program (Nichols, 1987).
ART	Automated Reasoning Tool. An expert system development environment by the Inference Corporation.
AV	Attribute-Value pairs. Facts with two elements representing the attribute name and its value of some (implicit) object.
BIM	Bus Interface Module. The VAMP module providing an interface between the PI-bus and an external bus.
BTB	Block Transfer Bus. The data bus in the avionics architecture designated for the high speed inter-processor transfer of large blocks of data.
CLIPS/Ada	The C Language Integrated Production System, implemented in Ada. A forward chaining expert system development system based on Rete (Melebeck, 1989a).
ES_Shell	The expert system shell designed for use on the VAMPs developed during this research.
HSDB	High Speed Data Bus. The fiber-optic high speed data bus developed to support inter-processor communications for the VAMPs (Common Module Systems, undated).
ITB	Integrated Test Bed. The integrated avionics testing facility of the Avionics Laboratory which contains a real-time man-in-the-loop aircraft simulation.
Join Net	The portion of the CLIPS discrimination network where pattern matches are joined together to activate a rule. This corresponds to the implicit "and" between LHS clauses of a rule.
Knowledge Base	The collection of rules and objects required to operate an expert system.

LHS	Left Hand Side. The antecedent or "if" part of an if-then rule.
LPU	Logical Processing Unit. The AOS designation for a collection of routines operate in the same logical address space on the VAMPs (Nichols, 1987).
MAB	Mission Avionics Bus. The data bus in the avionics architecture designated for inter-process message communication.
MID	Message Identification. The message identification number which the AARTS Operating System uses to route inter-task messages.
OAV	Object-Attribute-Value triple. Facts with three elements representing the name, attribute, and its value of some (explicit) object.
Pattern Net	The portion of the CLIPS discrimination net where incoming facts are matched against the constant tests of the LHS clauses of the compiled rule base.
PCA	VAX Performance and Coverage Analyzer. The VAX tool for analyzing software test coverage and program performance on a VMS VAX.
PES	Piloting Expert System. The aircraft piloting expert system knowledge base from the Robotic Air Vehicle project.
PI-bus	Processor Internal Bus. The intra-processor bus for communications between modules within the VAMP (Common Module Systems, undated).
RAV	Robotic Air Vehicle. A project to develop a system of distributed cooperating expert systems for control of an autonomous air vehicle (McNulty, 1987).
Rete	The pattern matching system consisting of a rule compiler and forward chaining inference engine developed by Dr Charles Forgy (Forgy, 1987).
RHS	Right Hand Side. The consequent or "then" clause of an if-then rule.
Rule Base	The collection of rules containing the expertise of an expert system.

SMM	System Mass Memory. A file-structured device on the High Speed Data Bus for the storage and retrieval of system files.
TIDIE	TI Dallas Inference Engine. The inference engine and knowledge representation system used on the Robotic Air Vehicle project (Lystaad, 1987).
V1750A	The VHSIC realization of the MIL-STD-1750A Instruction Set Architecture.
VAMP	VHSIC Avionic Modular Processor. The VHSIC implementation of the MIL-STD-1750A Instruction Set Architecture.
Working Memory	The "scratch pad" memory of the inference engine, containing objects and temporary bindings.

Appendix B. An Introduction to Production Systems and the Rete Algorithm (Fanning, 1988)

Although Artificial Intelligence (AI) encompasses many fields, including speech and vision processing, natural language understanding, and robotics, the general public's main exposure to AI has been through expert systems. Production systems, a particular type of expert system, are used in a variety of problem areas, such as fault diagnosis, autonomous vehicle navigation, route planning, and medical diagnosis. These systems are unique in that they are dealing with problems that:

a) require symbolic reasoning, rather than numerical computation,

b) require the use of heuristics, i.e. rules of thumb or default strategies,

c) require solutions based on uncertain or incomplete information,

d) require domain specific knowledge. (Prerau, 1985:27-28)

Problems with these characteristics have proven too computationally intensive and/or combinatorially explosive to be solved by traditional algorithmic methods. However, production systems, which were originally conceived by the mathematician Emil Post as a general computational model, (Valdez-Perez, 1986:32) work very successfully with these kinds of problems.

Production systems use a set of domain knowledge encoded rules to reason about the problem symbolically and to find a path from the initial state to a goal state. Production systems consist of a knowledge base and an inference engine. The knowledge base contains facts, which represent the current problem state stored in a "working" memory; and rules, which are IF-THEN structures representing the actions to be taken at specific problem states to achieve a goal state. The inference engine is a rule interpreter that

matches the IF part of the rules against the current problem state, and executes the THEN part of a matching rule. When more than one rule fits the current problem state, the inference engine uses a conflict resolution strategy to determine which rule or what order of rules to execute.

Some production systems use probabilistic methods to determine rule application when the current representation of the problem in working memory is either uncertain or incompletely defined. The knowledge base is constructed from problem domain-specific knowledge, modeled after the way in which a human expert solves problems, hence the name "Expert Systems." The process of acquiring the domain knowledge, developing the appropriate representation, and constructing the knowledge base is called Knowledge Engineering.

The Knowledge Base The knowledge base is composed of two parts, the fact base and the rules base. Each of these components is described below.

The Fact Base. In the problem space, an object of concern may have attributes such as color, size, or location. In the knowledge base, the values of these attributes are represented as facts, expressing assertions about properties, relations, and propositions, in the problem space. (Hayes-Roth, 1985:924)

Facts in the knowledge base are usually static, that is, they describe the problem state at a given time. Dynamic facts represent time-varying values and are placed in a "scratch pad" area of RAM called the working memory. The representation of the problem state in working memory is crucial to the efficient execution of the production system. Considerable care should be taken to devise a representation that is both understandable and easy to use. Early production systems represented facts as LISP lists, because LISP was designed to provide many functions for manipulating lists. (Valdez-Perez, 1986:31)

The Rule Base. The rule base is a collection of production rules. Each rule embodies a piece of problem domain knowledge concerning actions to be taken in specific problem states. Rules are most often written in an IF-THEN format, which is close to the human expert's natural way of solving problems. Figure 18 shows a production rule from a well-known production system example (Winston and Horn, 1984:271) for identifying animals.

```
(RULE IDENTIFY16

    (IF    ((> ANIMAL IS A (> TYPE))                    ;LHS
           ((< ANIMAL IS A PARENT OF (> CHILD)))

    (THEN  ((< CHILD IS A (< TYPE))))                   ;RHS
```

Figure 18. A Simple Inheritance Rule

In production rules, the IF part is commonly called the Left Hand Side (LHS), or antecedent clause; the THEN part is labelled the Right Hand Side (RHS), or the consequent clause. In this example, the rule is to be applied to any animal of specified type who is the declared parent of **CHILD**. The RHS specifies that if the antecedent is true (matched against working memory), then it is concluded that the **CHILD** is also the same **TYPE** as the parent, and that fact is placed in working memory (asserted).

The LHS determines the eligibility of the rule, and the RHS typically performs one or more of the following:

- Add to, delete from, or modify working memory,
- Perform I/O,
- Perform calculations on data extracted from matched working memory,
- Make calls to foreign language subroutines or the operating system. (Neiman and Martin, 1986:57)

While the rule in Figure 18 is designed to completely match against working memory, not all expert systems need complete knowledge of the problem in order to find a solution. Representations exist to deal with incomplete or uncertain data. The rules in MYCIN, a well-known medical diagnostic expert system, have probabilistic measures of uncertainty embedded in them that determine rule selection and execution. (Rich, 1983:287)

The order of rules in the rule base should not reflect the order of steps to be followed in solving the problem. The rule base should be unordered for two important reasons. First, each production rule should stand alone as a problem-solving step for a specific problem state; the programmer should not try to "group" rules into an execution sequence. Second, the inference engine usually tries to match all the rules against the current problem state and uses its own strategy to determine the execution order of the rules.

Expert System Types. Production systems are a particular type of expert system. Expert systems use three basic types of reasoning: forward chaining, backward chaining, and a combination of forward and backward chaining. Each will be discussed to show the differences between production systems and other expert systems.

Forward Chaining Systems. Forward chaining systems start from the current problem state and execute the rules whose LHSs are satisfied until either a goal state is reached or no more rules may be executed. This method builds a tree of solution paths,

starting with the initial state as the root node. The next level is constructed by matching the LHS of all rules against working memory and using the RHS of matched rules to generate new nodes (problem states). The tree continues to grow in this manner until a node matching the goal state has been generated or no more rules are applicable. (Rich, 1983:56)

Forward chaining is most effective when there are more possible goal states than initial states, and when the number of paths reachable from each node is small. For example, selecting and cooking a meal requires that the ingredients and means to prepare them exist before cooking actually begins. Since ingredients may be combined in different ways for various meals, the number of goals (eligible meals) is normally higher than the initial state (ingredients). Constraints placed on the combination of ingredients and their preparation tend to reduce the solution paths (candidate recipes). Therefore, forward chaining would be a good method for a robotic cook.

Backward Chaining. Backward chaining systems start from goal states and, using the rules, reason backward to the initial state. In this method, solution trees are also generated, with the goal state(s) as the root(s). Each level is generated by matching the RHS of the rules against working memory and using the LHS to generate new nodes. Reasoning proceeds in this manner backward until the initial state is reached. Since this process begins with the goal, this method is often called goal-directed reasoning. (Rich, 1983:57)

Backward chaining is most effective when there are more start than goal states, or more intermediary states that lead to a start state than goal states. For instance, when executing a route planner, it makes sense to back chain because there is only one goal: the destination, but many locations close to the start state. An advantage of backward chaining systems is their ease in providing justifications during the search process.

Forward and Backward Chaining Systems. Since most search techniques can be performed in either direction, some expert systems take advantage of both forward and backward chaining. One method is to reason both forward from the start state and backward from the goal state simultaneously. For rule-based systems, this approach can be ineffective. Searches proceeding in both directions could "miss" each other in the middle, causing extra search operations to be performed. Another method is a more iterative approach, where the direction of search is guided by some heuristic in the inference engine according to the benefits of a particular direction of search at a given problem state.

The Production System Inference Engine. By definition a production system is a forward chaining expert system, using IF-THEN structured rules, the so-called production rules. Simply viewed, it is a program that begins with a list of known facts, executes for a time, and eventually finishes when it can accumulate no new facts. (Valdez-Perez, 1986:30-31) The inference engine, also known as the rule interpreter, controls the execution of the program by matching the production rules' LHSs against working memory. A rule whose LHS has matched may have its RHS executed by the inference engine. This is called "firing" a rule. If more than one rule matches, it is the job of the inference engine to decide which rule to fire, a process known as conflict resolution.

A production system inference engine will execute the following steps in a loop until either a goal state is reached, or no new facts can be accumulated.

Step 1: MATCH Evaluate the LHS of the production rules to determine which are satisfied, given the current contents of memory.

Step 2: CONFLICT RESOLUTION Select one production rule with a satisfied LHS. If no production rules have satisfied, return control to the user.

Step 3: ACT Perform the actions specified in the RHS of the selected production rule.

Step 4: DONE? If a goal state is reached or no more rules match, return control to the user, otherwise go to step 1. (Forgy and Shepard, 1987:36)

The two desirable features of an inference engine are that it cause motion in working memory, and that it be systematic. (Rich, 1983:32-33) If an inference engine fires the same rule every time, a solution will probably not be reached. The same situation may occur if an inference engine fires rules without constructing a path toward a goal state. A system should satisfy the need for local motion (over the course of a single step), and global motion (over the course of several steps). (Rich, 1983:34)

Conflict resolution strategies vary according to the intent of the system. In systems where the optimal goal is not necessarily required, the simple strategy of choosing the rule with the most antecedent conditions could be used. A rule whose LHS contains many conditions is a) more specific than a rule with less conditions, b) will match fewer problem states, and c) is probably a more appropriate choice. (Forgy and Shepard, 1987:36)

A System Example. A simple production system will illustrate the basic concepts discussed thus far. The forward-chaining rule interpreter shown in Figure 19 is from the well-known animal identifier system in Winston & Horn's LISP. (Winston and Horn, 1984:275-283)

```
(DEFUN FORWARD-CHAIN ()                                    ;line 1
  (DO ((RULES-TO-TRY RULES (CDR RULES-TO-TRY))             ; 2
       (PROGRESS-MADE NIL))                                ; 3
      ((NULL RULES-TO-TRY) PROGRESS-MADE)                  ; 4
      (COND ((USE-RULE (CAR RULES-TO-TRY))                 ; 5
             (SETQ RULES-TO-TRY RULES)                     ; 6
             (SETQ PROGRESS-MADE T)))))                    ; 7
```

Figure 19. A Simple Rule Interpreter

The function **FORWARD-CHAIN** acts as the production system inference engine. The **DO** form on line 2 initially binds the variable **RULES-TO-TRY** to the list of **RULES**. Typical production rules in LISP form are shown in Figure 20. On each successive run through the **DO** loop, **RULES-TO-TRY** is bound to the CDR of **RULES-TO-TRY**, that is, if **RULES-TO-TRY** is thought of as a stack of rules, then the top rule pops off the stack on each iteration.

The **COND** form (lines 5-8) performs the pattern matching by calling **USE-RULE**, which matches the LHS side of the current rule against working memory. Conflict resolution in this example is simple because **USE-RULE** also executes a rule's RHS as soon as it matches. If a rule is fired, the **USE-RULE** expression (line 5) tests positive, and lines 6 and 7 are executed. Line 6 resets the list of **RULES-TO-TRY** to the entire set of **RULES** again. Line 7 sets the **PROGRESS-MADE** variable to **T** (true), signifying a rule has been fired. If the rule does not fire, the **USE-RULE** form tests negative, the rule is popped off the list of **RULES-TO-TRY** (line 2), and the loop is repeated. The test for

completion is when **RULES-TO-TRY** becomes empty, i.e. none of the rules fired in the current problem state. At that time, the value of **PROGRESS-MADE** is returned.

Some sample rules are shown in Figure 20. The rules are implemented as LISP lists. The pattern matcher keys on the tokens **IF** and **THEN** to identify the LHS and RHS of the rule, respectively. The "**>**" token identifies a pattern to be matched and "pulled in" from working memory (e.g. **>ANIMAL**), and the "**<**" signifies that the previously matched value should be used.

Working memory variables (facts) are also represented as LISP lists. The lists are stored in the list **ASSERTIONS**. Figure 21 is a list of sample assertions about the animals **ROBBIE** and **SUZIE**. Executing the system with the rules shown in Figure 20 and the facts in Figure 21 would produce the output shown in Figure 22.

The first rule to be matched against the facts is rule **IDENTIFY1**, which matches against line 5 of Figure 21 "**ROBBIE HAS HAIR.**" The consequent "**ROBBIE IS A MAMMAL**" is placed on the front of the list of assertions. Since **IDENTIFY1** doesn't match any more facts, no more assertions are made by this rule, and the system moves on to rule **IDENTIFY3**. The LHS of this rule matched the assertion "**SUZIE HAS FEATHERS**" on line 6 of Figure 21. Matching and asserting proceeds in like manner until the last assertion can be made (the last rule fails to match), and the systems halts execution. Notice that the antecedent condition "**<ANIMAL HAS BLACK STRIPES**" keeps rule **IDENTIFY10** from firing, while the more correct rule **IDENTIFY9** correctly identifies **ROBBIE** as a **CHEETAH**. This demonstrates the fact that all the antecedent conditions must

be met before the consequent actions can be taken. This is also the case for rules **IDENTIFY14** and **IDENTIFY15**.

```
(RULE IDENTIFY1
      (IF    ((> ANIMAL) HAS HAIR))
      (THEN ((< ANIMAL) IS MAMMAL)))

(RULE IDENTIFY3
      (IF    ((> ANIMAL) HAS FEATHERS)
      (THEN ((< ANIMAL) IS BIRD)))

(RULE IDENTIFY5
      (IF    ((> ANIMAL) EATS MEAT))
      (THEN ((< ANIMAL) IS CARNIVORE)))

(RULE IDENTIFY9
      (IF    ((> ANIMAL) IS MAMMAL)
             ((< ANIMAL) IS CARNIVORE)
             ((< ANIMAL) HAS TAWNY COLOR)
             ((< ANIMAL) HAS DARK SPOTS))
      (THEN ((< ANIMAL) IS CHEETAH)))

(RULE IDENTIFY10
      (IF    ((> ANIMAL) IS MAMMAL)
             ((< ANIMAL) IS CARNIVORE)
             ((< ANIMAL) HAS TAWNY COLOR)
             ((< ANIMAL) HAS BLACK STRIPES))
      (THEN ((< ANIMAL) IS TIGER)))

(RULE IDENTIFY14
      (IF    ((> ANIMAL) IS BIRD)
             ((< ANIMAL) DOES NOT FLY)
             ((< ANIMAL) SWIMS)
             ((< ANIMAL) IS BLACK AND WHITE))
      (THEN ((< ANIMAL) IS PENGUIN)))

(RULE IDENTIFY15
      (IF    ((> ANIMAL) IS BIRD)
             ((< ANIMAL) FLIES WELL))
      (THEN ((< ANIMAL) IS ALBATROSS)))
```

Figure 20. Sample Rules for the Animal World (Winston and Horn :275-283)

```
(SETQ ASSERTIONS                    ; 1
      '((ROBBIE HAS DARK SPOTS)     ; 2
        (ROBBY HAS TAWNY COLOR)     ; 3
        (ROBBIE EATS MEAT)          ; 4
        (ROBBIE HAS HAIR)           ; 5
        (SUZIE HAS FEATHERS)        ; 6
        (SUZIE FLIES WELL)))        ; 7
```

Figure 21. Sample Assertions for the Animal World

```
(RULE IDENTIFY1 SAYS ROBBIE IS A MAMMAL)     ; 1
(RULE IDENTIFY3 SAYS SUZIE IS A BIRD)        ; 2
(RULE IDENTIFY5 SAYS ROBBIE IS A CARNIVORE)  ; 3
(RULE IDENTIFY9 SAYS ROBBIE IS A CHEETAH)    ; 4
(RULE IDENTIFY15 SAYS SUZIE IS A ALBATROSS)  ; 5
```

Figure 22. Sample Output of the Animal Identifier

This simple example shows some flaws of simple production systems. The first is a lack of clear goal states. When any rule fires, a goal is apparently achieved, but the ultimate goal of this system is the most descriptive classification of animals possible. There is no way to discern which of these statements is more correct: "**ROBBIE IS A MAMMAL**", or "**ROBBIE IS A CHEETAH**." Both are correct, but "**ROBBIE IS A CHEETAH**" is a stronger assertion. A second flaw is that the order of firing of the rules is always front-to-back. So, **RULES-TO-TRY** forces an implicit execution sequence on the rules. A programmer could order the rules so as to grossly affect the results. This would violate the abstract boundary between the rules and the inference engine by embedding a control strategy in the structure of the rule base. A third flaw is that nothing prevents a rule from repeatedly firing. Because this system always adds new assertions to the front of the list.

and each rule is matched against the most recent data, if the RHS of a rule caused little change in working memory, a solution might never be reached.

While this simple example is not the perfect production system, it does demonstrate the basic concepts and execution. It also shows the need for some improvements including maintaining some history of the rules that asserted facts into working memory (used for an "explanation" facility); providing a better conflict resolution for choosing between multiple rule matches; matching only against facts which have changed. not the entire fact base; and eliminating rules that duplicate previous assertions. The issues of better conflict resolution strategy, smaller matching scope, and elimination of duplication have been addressed by the Rete Algorithm, described in the next section.

The Rete Algorithm The Rete Match Algorithm is a fast method for comparing a set of patterns to a set of objects to determine all the possible matches. (Forgy and Shepard, 1987:34) Developed by Charles L. Forgy in 1974 at Carnegie Mellon University, Rete was designed to handle the unusually large pattern-matching operations in production systems. Rete has several advantages over its predecessors. It greatly reduces the number of redundant matching operations, provides a good scheme for conflict resolution, and is *fast*. Rete is touted as the most efficient algorithm yet developed for performing match operations on a single processor.

The Rete Algorithm gains most of its speed both by limiting the scope of its matching, and by eliminating redundant antecedent conditions. For small production systems with few rules and objects, pattern matching is accomplished by matching all rules against all objects in working memory. Real-world problem solving often involves hundreds of objects, each consisting of 10 to 100 attributes, and hundreds, or even thousands of rules. Continuing the all rules/all objects matching scheme would be

extremely inefficient. In fact, much less than 1% of working memory changes on each typical cycle, and usually no more than two to four changes are made by each rule firing. (Forgy and Shepard, 1987:37) Rete takes advantage of this by saving information about LHS clause matches, updating the information when more changes in working memory are made. Matching information is stored from cycle to cycle, sharply reducing the number of new matches needed. In this way, the speed of the inference engine will depend on the rate of changes in working memory, not on the size of working memory. This reduces matching from an exponential function to a nearly linear function.

In a typical production system, many of the antecedent conditions of rules are identical. Since all rules access the same objects in working memory, logically many of the conditions in their IF parts will be similar or identical. Rete uses this concept by scanning the rule base before run-time, and compiling the rules into a data-flow graph. Rete combines the identical IF parts into a single nodes on the net and links the nodes to all the places they are used. In this graph, called the Rete Net, nodes in the graph contain the operations to be performed by the matcher. The links represent the way data is to flow from one node to another in the graph. Redundant computations are eliminated by constructing only a single node to perform a given match operation, linking the node to places where a result is needed. (Forgy and Shepard, 1987:37)

The Rete Net. The Rete Net serves as a function to map changes in working memory into changes in the set of satisfied production rules. (Forgy and Shepard, 1987:37) Changes in working memory are passed through the net as tokens. When a token is received by a node, the node processes it according to the match computation stored there, and, if necessary, passes the token along to the next node. When a token has passed all the way through the net, then a rule has "matched,"and that rule is added to the conflict resolution set.

There are basically two kinds of data tokens. Alpha Tokens are ordered pairs. The first piece of information is a tag, a pointer to an object in working memory. The second piece is an indication to the nodes whether the element is being added to or deleted from working memory. An Alpha Token is intended for single pieces of data. Beta Tokens are ordered lists. They consist of two or more data tags, and an indication of whether to add or delete the objects.

There are basically four kinds of nodes in a simple Rete Net: root nodes, single input nodes, two-input nodes, and terminal nodes. A root node performs no processing and is used mainly as a starting point for the system. A data token received by the root node is immediately passed along to its successor nodes. A single input node will test a particular value of the object in an alpha token. If the test succeeds,the node makes a note of the match and passes an identical copy of the token to its successor nodes. A two-input node is used to tie two single input nodes together. The two-input node keeps track of successful tests by its predecessor single input nodes, and determines when conditions are met by the tokens that satisfy both input conditions. For a rule with N conditions in its IF clause, N-1 two-input nodes will be generated to tie them together. The terminal node adds the completely satisfied rule to the conflict set. The terminal node contains information about the antecedent conditions of a rule and about when it received a data token, but all the elements referred to by the token must satisfy the terminal node before it will nominate a rule for firing.

Construction of the Rete Net is a two-step process. First, objects are compiled into working memory classes, with associated attribute slots. In OPS5, an early implementation of Rete, attributes are stored on LISP property lists. Objects with the same attributes share attribute slots. Second, the production rules are compiled into the Rete Net. (Neiman, 1987:43)

An example of a one-input node rule is **IDENTIFY1** from Figure 20. The antecedent clause consists of a single condition and would compile into a single input node net as shown in Figure 23.

If the statement **(ROBBIE HAS HAIR)** is asserted in working memory, a token will be passed to the first one-input node, which will match the **>ANIMAL** with **ROBBIE** and pass the token on to the next node. The next node will match the **HAS** condition with the value **HAIR**. Passing this test, the node will pass the token on to the terminal node. Determining that the LHS of **IDENTIFY1** has been completely satisfied, the terminal node will place the RHS of the rule, the assertion that **(ROBBIE IS MAMMAL)** on the agenda to await execution.

```
(RULE  IDENTIFY1
       (IF    ((> ANIMAL) HAS HAIR))          ;LHS
        (THEN ((< ANIMAL) IS MAMMAL)))        ;RHS
```

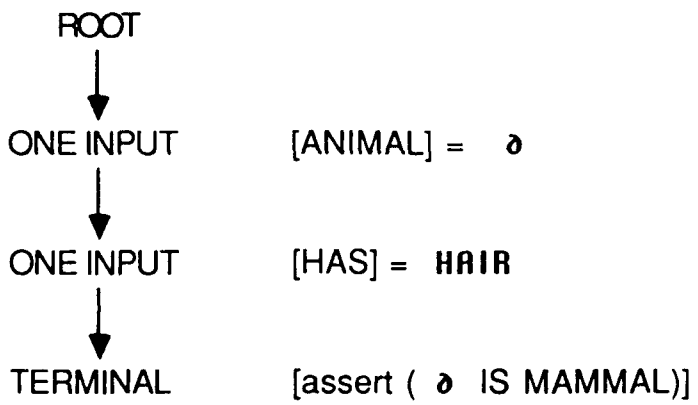

Figure 23. A Sample Rule Using One-Input Nodes

A rule from a blocks-world system that would be compiled using a two-input node is shown in Figure 24.

```
(RULE FIND-COLORED-BLOCK
      (IF   ((> GOAL) IS (FIND-COLLORED-BLOCK (> COLOR))
            ((> BLOCK) COLOR IS (< COLOR)))
      (THEN ((< GOAL) IS SATISFIED)
            (FOUNDBLOCK IS (< BLOCK))))
```

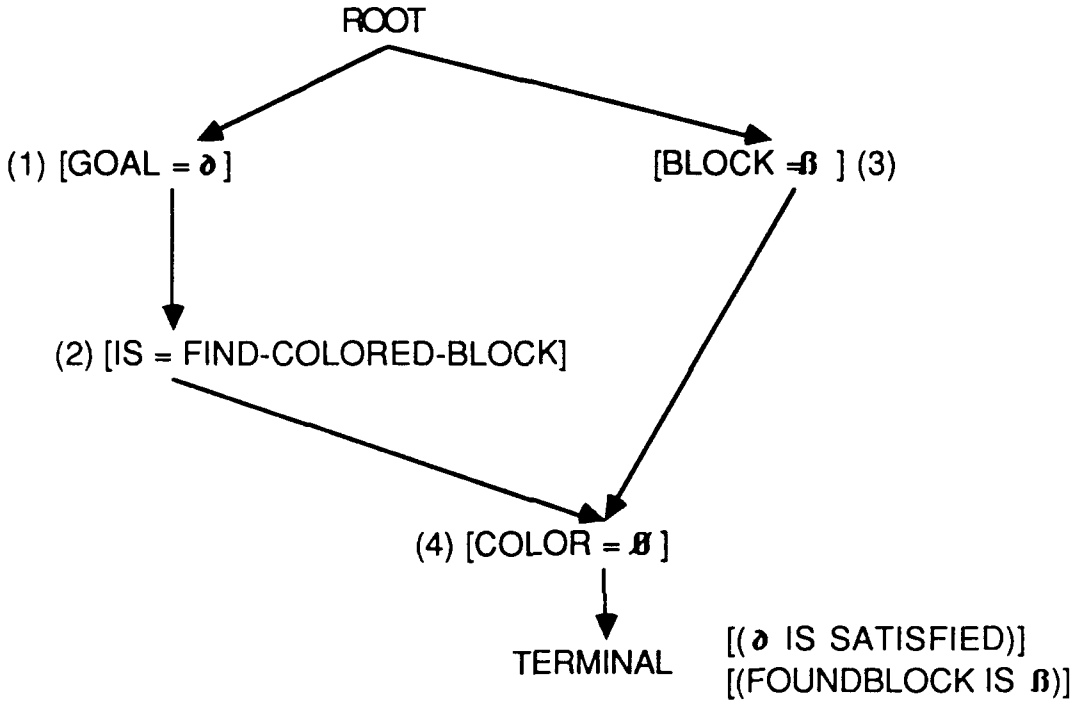

Figure 24. A Sample Rule Using a Two-Input Node.

When the root node in this example receives a data token, it passes into both successor nodes. Nodes 1, 2, and 3 are single input nodes, while node 4 is a two-input node. Node 1 will try to match on tokens referring to a **GOAL**. Such a token will be passed on to node 2 to determine if the current goal is to find a colored block. Node 3 will try to find a block in each token. Both nodes 2 and 3 will pass their matched tokens on to node 4. When node 4 has received successful tokens from both its predecessors, it will send a token to the terminal node. If the assertions **(GOAL42 IS (FIND-COLORED-BLOCK RED))** and

(BLOCK31 IS RED) exist in working memory, the terminal node will place the clause to assert **(GOAL42 IS SATISFIED)** and **(FOUNDBLOCK IS BLOCK31)** in the conflict set. For more advanced implementations of Rete, there are many other kinds of nodes and token operators, but the nodes and tokens presented here are the most frequently used.

Conflict Resolution. The real function of the Rete Algorithm is to produce the conflict set (the agenda). The conflict set is the collection of rules whose LHS clauses have matched and are waiting to have their RHS executed. Since Rete-based systems place a high priority on conflict resolution, their inference engines operate with a modified production order. The typical order, as stated previously, is (1) MATCH, (2) CONFLICT RESOLUTION, (3) ACT, and (4) QUIT or GOTO (1). The modified production order for Rete is:

Step 1: CONFLICT RESOLUTION Input is the conflict set (the output of Rete). Select one rule with a satisfied LHS. If no rules have a satisfied LHS, QUIT

Step 2: ACT Perform the actions specified in the RHS of the selected rule

Step 3: MATCH Evaluate the LHS of the rules to determine which are satisfied given the changes in working memory from step 3. Output is the conflict set

Step 4: QUIT if no rules match, otherwise go to step 1.

The conflict set is a list of ordered pairs. Each ordered pair consists of the rule name, and a list of elements matched by that rule's LHS. Rete conflict resolution is finding one ordered pair from the set that dominates all others. One conflict resolution strategy is called MEA.

MEA ensures that

- A rule will fire only once upon the assertion of a working memory element,
- The most specific rule (judged by the number of LHS conditions) will take precedence,
- The most recently added working memory elements will be most likely to cause a rule to fire.

This strategy is suitable in most cases, but has some limitations. There is no mechanism to allow all or a number of eligible rules to fire, if a programmer wishes to bypass conflict resolution for a block of tasks. However, since MEA tends toward the latest data (and therefore, the most recently fired rule), it is easy to add a rule to an existing rule base and have that rule fire at the right time.

Applications of the Rete Algorithm. The Rete Algorithm is currently being used in a wide variety of commercial applications and development tools. It has been implemented in LISP, BLISS, Pascal, and C. All of the OPS systems are production systems that use the Rete Algorithm. OPS5 was the first, implemented in LISP. OPS83 is implemented in C.

OPS83 is of special note, because it has been specially modified for efficient operation. OPS83 programs are linked to the C run-time system/libraries. (Neiman and Martin, 1986:68) Operation on a VAX is estimated at thirty times faster than the LISP implementation of OPS5. To achieve this speed, the OPS83 compiler "walks over" the compiled Rete Net, and writes out machine code at each node to do the specified match operation. The computer hardware then actually becomes the system interpreter. (Forgy and Shepard, 1987:39)

Commercially available production system development tools include Inference Corporation's Automated Reasoning Tool (ART) and the Carnegie Group's Knowledge Craft for LISP machines, Gold Hill's GoldWorks for the IBM AT, and ExperIntelligence's ExperOPS5 for the Apple Macintosh. These production system "shells" are useful for problems requiring greater efficiency for real-world problem solvers like Digital Equipment Corporation's R1/XCON, and Texas Instruments' Robotic Air Vehicle.

Summary Production systems are increasingly being used to solve problems that cannot be easily or quickly solved by traditional algorithmic computer methods. Since production systems can reason about symbols, objects, and ideas, they can use problem domain-specific knowledge in the form of rules to solve a problem forward from the current state to a goal state. The Rete Algorithm is an efficient way of dealing with problems that may be solved by production systems where there are many rules to be applied, and many objects to consider. Implementations of the Rete Algorithm exist in higher order languages for popular classes of computers, and production systems and system shells based on Rete are now being used in commercial applications to solve real-world problems.

Appendix C. ES_Shell Implementation Notes

This appendix compliments the design discussion presented in Chapter Three, and the ES_Shell source code. The implementation of the ES_Shell on the VAX with the AOS Shell and testing with the PES rules are described informally. The challenges to the implementation are presented along with their working solutions. This appendix is intended for programmers who desire a better understanding of why certain approaches were taken in the code. Readers desiring only a high level understanding of the ES_Shell design will find this in Chapter Three.

The ES_Shell and the AOS. This project grew from work in two areas: embedded flight computer operating system design, and the application of expert system to embedded avionics control. The Avionics Laboratory, Wright Research and Development Center, while developing the kernel operating system for the next generation flight computer, envisioned an expert system shell as a part of this operating system. The expert system shell would function much like a system service, such as file I/O or message passing routines. The operating system was to be developed in Ada and operate on an advanced flight processor architecture. The lab had sponsored research into expert system control of air vehicles, and the product of this research seemed a natural application to test with the expert system shell running under the new operating system.

The Ada Avionics Real-Time Software (AARTS) Operating System (AOS) provides the kernel operating system for the VHSIC Avionic Modular Processor (VAMP), the next generation of the Mil-STD-1750A instruction set architecture. The target application is the Piloting Expert System (PES) of the Robotic Air Vehicle (RAV) project which contains the piloting knowledge necessary to pilot an autonomous Air Vehicle on a low-level

reconnaissance mission. The PES currently executes in a forward chaining expert system shell on a LISP machine, passing messages to distributed system of cooperating expert systems in the RAV. The expert system shell, written in LISP, is based on Rete and tailored for ES development, debugging and testing on the PES implementation, but an Ada-based expert system shell had to be developed for use under AARTS.

The development of the new expert system shell, called ES_Shell, proceeded according to the following steps:

- Selecting the inference engine components necessary to support the ES_Shell,
- Selecting the AOS services necessary to accomplish the ES actions,
- Determining the interfaces between the inference engine and the problem environment,
- Developing the ES_Shell in VAX Ada,
- Testing the ES_Shell with the AOS Shell on a VAX,
- Testing the PES running on the ES_Shell and simulated aircraft model,
- Evaluating ES_Shell's performance, and
- Recompiling ES_Shell with TARTAN Ada and the AOS and testing on the VAMPs.

Development of the ES_Shell. The ES_Shell provides the interface between the inference engine and the problem environment-- the real-time man-in-the-loop aircraft simulation used for avionics testing in the Avionics Lab's Integrated Test Bed (ITB) facility. The ES_Shell would manage requests from the ITB's Operational Flight Program (OFP) for expert system services which would include system initialization, data update, and problem solution. The ES_Shell would also provide interfaces for the inference engine to interact

with the simulation, either to control it or communicate with other flight software, such as automated terrain followers or route planners.

The design of the ES_Shell is described in Chapter three of this thesis. An abbreviated module specification is given below.

ES_Shell

Inputs: Procedure or task entry.

Processing: Performs the typical expert system cycle: update, match, select, and execute. Acts as the main interface to the AOS. Defines messages, objects, and events.

Outputs: None

LOAD

Inputs: Rule base filename

Processing: Opens rule file, loads rules, parses them for correct syntax, compiles correct rules into Rete net. May load object definitions from file, creating entries for objects in working memory.

Outputs: Rete net

UPDATE

Inputs: Incoming bus messages

Processing: Performs update, and by assertion, implicitly the match portion of the expert system cycle. Provides interference to incoming bus messages and AOS message services. Filters data and asserts object changes to Rete net.

Outputs: Assertions to Rete net, changes in rule activations to agenda.

FORWARD CHAIN

Inputs: Agenda (list of activated rules in Rete net)

Processing: Performs select and execute portions of cycle. Selects a rule from agenda and executes its RHS. New data may be retracted or asserted as consequence of RHS actions in turn causing changes to agenda. Continues to select and execute rules until agenda is empty. Provides interface between inference engine and problem environment via user-defined functions, including outgoing messages.

Outputs: Modifications to Rete net. Outgoing bus messages. Returns number of rules fired to calling program.

CLIPS/Ada Components. ES_Shell required a forward-chaining inference engine based on Rete and written in Ada, and the means to access and modify elements in the inference engine's working memory. A Rete based system uses a compiled network representation of the production rule base, so some means of loading and compiling the rule base was also required. Since there are already inference engines written in Ada which are based on Rete, the development of the inference engine itself was not a part of this project.

CLIPS/Ada is a Rete-based expert system development tool distributed by NASA. It is an Ada implementation of the C Language Integrated Production System (CLIPS). CLIPS/Ada is available at no cost to government personnel and contractors by calling the CLIPS users help desk at (713) 280-2233. The components used from CLIPS/Ada to support ES_Shell were the rule/loader and compiler, the assert and retract mechanism, and the CLIPS inference engine.

The rule loader/compiler begins by opening the production rule file, and then reads the production rules in one at a time. As the rules are loaded, their syntax is pursued for correctness. Rules with valid syntax are compiled into the Rete net; others are discarded.

The routine compiles the net to take advantage of the structural similarity of the rule's LHS clauses. LHS clauses which are common to multiple rules are shared in the net as object pattern tests. This sharing allows some space savings in the net.

The inference engine uses the agenda produced by the asserts and retracts, selecting the rule at the top of the list for execution. The RHS actions of the rule referenced by a pointer from the agenda are executed in turn. Each RHS action may further modify working memory, producing a new agenda. The inference engine continues to run until the agenda is empty, or a run limit set by the calling program has been reached.

ES_Shell UPDATE Module. The ES_Shell also performs the update part of the typical production system cycle of update, match, select, and execute. During the update, data is brought in from the simulations' aircraft communications bus, decoded, and filtered for novel data. Changes in the problem state are introduced into the inference engine's working memory by the ES_Shell.

In order to introduce these changes into working memory, assert and retract mechanisms were required. The CLIPS ADD_FACT and RETRACT routines were chosen to fulfill this need.

Facts are prepared for the assert by determining the fact length, allocating the fact memory segments, typing the fact segments, and inserting the data values appropriately. The CLIPS UTILITY package provides the necessary routines, in GET-Segment, Set-Segment-Kind, and Set-Segment-Value. The method of preparing facts using these functions is described in the CLIPS/Ada Advanced Programming Guide.

Once the fact has been prepared, it may be asserted using the ADD_FACT routine. It is important to note that the ADD_FACT function scans the list of known facts for duplication. In the event of a duplicate, the new fact is discarded, and its storage space deallocated. No notice of this is given to the calling program by ADD_FACT. Further use

of the pointer to the discarded fact may cause system error, so care must be taken to avoid this situation.

The RETRACT function requires a pointer to reference the fact to be removed from working memory. The UPDATE Module of ES_Shell must therefore maintain pointers to all facts which it asserts as each fact may eventually be retracted.

Many objects in the piloting domain are well-defined, and attributes of these objects, such as airspeed and altitude are known *a priori*. It was convenient to pre-allocate pointers to facts representing these attributes. The pointers are stored in an array indexed on the attribute name.

CLIPS has no mechanism for modifying a working memory element in place, so for each fact update, the old fact had to be retracted, and a new fact representing the update created and asserted. This introduced additional overhead in the extra deallocation and reallocation of fact memory, and possible consequent activation and deactivation of rules.

AOS Components. The AOS services necessary to support the ES_Shell were the MESSAGE_IO, TIME, EVENT, and FILE_IO. FILE_IO was required to load the rule file from the System Mass Memory (SMM) device. MESSAGE_IO was used for communications between the inference engine and other parts of the simulation. The EVENT services were used for task synchronization and to provide a rough interrupt capability. The TIME services were used to evaluate the ES_Shell's execution performance at run time.

CLIPS provides an interface to the inference engine which may be used by external routines to call for execution. This interface was used by ES_Shell to perform the select and execute parts of the production system cycle. In order to control the simulation from a rule's RHS, the ES_Shell was required to relay RHS actions to the environment. The CLIPS inference engine has the capability to execute user-defined functions on the RHS of

a rule. Using this capability, a user may define a function name and augment list which become a part of the RHS syntax. The function SEND was developed to accommodate RHS control of the simulation. It takes as augments the simulation object to be modified, and its new value.

The UPDATE and SEND routines described earlier use the AOS MESSAGE _IO services to receive and transmit messages over the simulation's communications buses. UPDATE receives aircraft state vectors and other environment information via the bus. Bus message formats and ID numbers must be agreed upon by both sender and receiver at compile time. It was convenient to structure the incoming data buffers as Ada records where the record component names correspond to aircraft attributes such as airspeed or altitude. The component names were also defined as an enumerated range which was used to index into the array of pointers whose facts represented the attributes. For instance Data Buffer AIRSPEED correlates to FACT_ARRAY(AIRSPEED). Use of this convention made the code more readable, and the fact array easier to transverse, aiding in the domain independence of the update algorithm.

To reduce the probability of duplicate fact assertion, UPDATE filters the incoming data. Filter values specify the minimum amount of change in a data value before it is reflected in the net. These filter values are domain specific and must be determined by the expert system developer prior to run time.

This approach requires the storage of the last update value. The values in the CLIPS fact list could be used, but their retrieval is slow and cumbersome compared to a simple record retrieval. A record structure was defined, then, to store the last update values. Both the old value record and the filter value records use the same record structure as the new data record. This makes the filtering code easier to read and maintain. An example filter is shown in Figure 25.

The SEND function transmits messages from the inference engine to the external environment. During the development of ES_Shell, outgoing messages were limited to those going to the aircraft simulation, but SEND could be used to communicate with other applications running on other CPUs or in other VAMPs. The SEND function receives its arguments from the inference engine. The first argument is a message ID number which corresponds to the logical address of the receiver. Following the ID number is an argument list containing the pieces of information to be transmitted. The message ID is required because the AOS MESSAGE_IO services use message connections similar to pipelines during communications. The SEND function must reference the connections explicitly for each distinct message, and so must contain separate processing to encode and transmit each message.

```
if abs(AIRSPEED.NEW - AIRSPEED.OLD) < AIRSPEED.FILTER then
    AIRSPEED.OLD := AIRSPEED.NEW;
    MODIFY(FACT_ARRAY(AIRSPEED),AIRSPEED.NEW);
end if;
```

Figure 25. A Sample Data Filter

Implementation Challenges. Several problems in the CLIPS/Ada and AOS code had to be overcome before the ES_Shell could be effectively tested. These problems and their working solutions are described in this section.

CLIPS Bugs.

Symbol Reference Counts. When a fact is created and a fact segment is used to represent a string or word, the string or word is hashed into a symbol table, and a pointer to the symbol table entry is stored in the fact segment. Several facts may reference the same symbol, which saves storage space if the symbol requires more space than the pointer. As each new fact references the symbol, the symbol's reference counter is incremented. Similarly, as each of these facts is retracted, the reference counter is decremented appropriately. Prior to retracting the old fact during the update process, the fact segment values are retrieved. The CLIPS routines which retrieve these values return a pointer if the value is a symbol. The retrieved symbol pointers are assigned to the new fact. If the retracted fact , however, was the last to reference these symbols, their counters are decremented to zero and the symbols' storage is deallocated. The newly created fact then contains pointers to deallocated space, which often produced undesirable results. To work around this, when the newly created facts segment have been assigned the symbols, their symbols' reference counters were explicitly incremented using the symbol package routines provided for that purpose. After asserting the fact (which automatically increments the counters) the counters were explicitly decremented. The usual order of retract before assert was maintained to reduce conflicts in rule activation and deactivation.

Memory Allocation. Another problem was caused by CLIPS memory segment allocation routines. When storage is set aside for a new fact, the memory is allocated internally by CLIPS from a temporary free space list. Presumably, when the memory segments are allocated or asserted, this temporary list should be updated. The temporary list is used by the inference engine for the creation of temporary variable bindings. These temporary bindings are deallocated after the inference cycle has completed. When the inference engine performed this deallocation, the newly created facts were

deallocated as well, also producing undesirable results. To work around this, the temporary allocation list was explicitly reset after each fact creation.

Working Memory Modification. The way the CLIPS functions handle working memory caused some problems in the development of the ES_Shell. When a fact is initially asserted, a pointer is returned which references its entry in the fact list. In order to externally retract the fact, this pointer must be known. Therefore, the external routine must keep the fact pointer in order to successfully modify working memory.

If the fact is modified internally, say as the consequence of a RHS action, the pointer to the new fact will change. The pointer maintained by the external routine may then be referencing a deallocated memory segment. Any further attempt by the external routine to access this fact may result in program termination, or worse, return garbage memory.

To get around this working memory was partitioned between objects which could be accessed and modified only by external routines and those objects which could only be modified internally. This type of partitioning was arbitrary and caused extra work for the inference engine, and affected the way rules were written. This partitioning is not checked by the parser, making the programmer responsible for enforcement. User enforcement of this kind of problem is clearly undesirable in an embedded avionics environment. An approach which allows unrestricted internal and external access and modification of working memory is needed.

Error Handling. CLIPS error handling and reporting caused some problems in the development, as well. In some cases, lower level functions are operating many levels below their caller. Multiple routines are often serviced by a single error handler, which may, in turn, pass errors up the call chain. By the time an error reaches the caller, it may be ambiguous or misleading. CLIPS makes heavy use of status variables at lower

levels and exceptions at higher levels. A more consistent approach should be used. Exceptions are the preferred Ada approach, but are often harder to service and then resume program flow.

AOS Bugs.

Asynchronous Messages. The AOS MESSAGE_IO's handling of asynchronous messages caused some problems during the development of the ES_Shell. MESSAGE _IO leaves the control of the message reception buffer up to the receiving program. If the receiver locks the buffer while processing it, incoming messages may be lost. The aircraft control messages contained in the RHS actions of the PES are asynchronous in nature, and losing any of these messages was considered intolerable. A two-way software handshake was instituted to restrict the sender from transmitting until the receiver is ready to process. The effect of this scheme on the inference engine is determined by the amount of time it must wait for the ready handshake. Some kind of message buffering or better handling of asynchronous messages by the AOS should be researched to ensure the inference engine doesn't have to wait indefinitely for a receive to become ready.

File Services. Another problem lay in the AOS FILE_IO services. The SMM is a byte-oriented file structured device which exists on the HSDB for storage and retrieval of information. FILE_IO provides all of the basic file services to the SMM such as file open, close, read, and write operations. Because the SMM is byte-oriented and the CLIPS rule loader is string or line oriented, modifications would have to be made to either FILE_IO or the CLIPS loader to accommodate rule loading and compiling. The physical interface between the SMM and the VAMPs was not operational during the period of this research, so modifications to the AOS FILE_IO or the rule loader were deferred.

The question was raised, however, whether the rule loader should be a part of the run-time ES_Shell at all. The loading and compiling process takes place only once, presumably at system initialization. If the ES_Shell were to truly support calls inflight, it might be necessary to load and compile new rule files on the fly. If the rule bases are considered to be static in flight , a view heartily supported by the avionics community, then all rule base compilation could be performed prior to the flight. Now a method would be required to load only the pre-compiled nets into memory. This eliminates the need for the rule parser and compiler, saving space in the code segment of the ES_Shell.

The subject was discussed with COSMIC, NASA's CLIPS support organization, who agreed to provide a method for accomplishing pre-run compilation. Their approach involves compiling the rules into a net, and saving the Ada representation of the net out to a set of package specifications. These specifications would be compiled and linked with the ES_Shell and associated with the inference engine at run time. This approach could allow multiple compiled nets to be linked with ES_Shell and be referenced in context by the use of pointers. The net pre-compilation procedure provided by NASA was suitable for testing with a single small rule base and was used during development.

Two issues have yet to be resolved before this method may be used successfully. The first is that while parsing and compiling at run time helps to realize a code segment reduction, the linking of multiple nets to the system increases the size of the data segment. As the results in Chapter Five show, the increase in space for a compiled net is significant and easily exceeds the VAMP's 64 Kword data segment limit. An approach which considers actively swapping compiled nets in and out of processor memory needs to be considered.

The second issue is of truth maintenance. In a real-time non-monotonic environment, data potentially may change at a rapid pace. How does one account for the

current problem state when a new net is brought into context? Simply reasserting all facts in the fact list may be unnecessary and require a large overhead. It may be possible to partition objects in the problem environment by context but this partitioning may be difficult as some objects may span many inferencing contexts. This difficult issue was not addressed in the development of the ES_Shell and requires further research.

Implementation Effects on ES_Shell Design Goals. Some of the development goals of ES_Shell were to make it modular in design and interruptable, as OFP application tasks must often be halted in the midst of their execution. To operate in a real-time environment, the ES_Shell must be able to support real-time requirements. For ES_Shell to be considered useful for further development, it should also exhibit independence from the problem domain as much as possible.

The ES_Shell was designed into three main modules - the rule loader/compiler, the data update module, and the inference engine. The use of Ada in the development of ES_Shell facilitated the modular approach. Each module was defined as a package. The individual modules included the CLIPS and AOS services necessary in the context of their own packages. The modular approach made testing and debugging the modules individually easier as well.

Interruptabilty. Interruption of an inference engine during operation can have disastrous results if the engine is modifying working memory. The Rete net may be left in an indeterminant state if some operations are not complete.

Some operations, therefore, must be considered atomic, and hence not interruptable. For the purposes of this research, the assert and retract mechanisms must be considered as such. The routines may not be interrupted from the time they enter the Rete net to the time when they exit and rules are activated or deactivated accordingly. In a larger sense, the

logical MODIFY operation, which consists of a retract followed by an assert should be viewed in much the same way.

In order to accomplish this approach, an interrupt check would have to be made at the top of the inference engine's select-execute-match cycle. Since minimal modifications were desired on the CLIPS software, the check was placed in the outer control loop in the ES_Shell itself. Consequently, this placement restricts the interrupt point to the end of the inference chain (i.e. when the inference engine runs out of rules to fire). The time, then from the interrupt signal to the halt in ES_Shell depends on the state of working memory, and the rules active in the inference chain.

A rough task interruption was implemented using the AOS EVENT services. The inference engine executes the match-select-execute cycle until it turns out of rules to fire. This constitutes the inner loop of the ES_Shell. The outer loop consists of the processing of new data, and calling the inference engine. To minimize the impact on the CLIPS code, a check of the EVENT ES-HALT was placed at the top of the outer control loop. If this event is set ES_Shell will exit the outer loop. This, of course, will only take place after all eligible rules have been fired in the inner loop. If a finer grain control of the ES_Shell is desired, the event check may be placed at the top of the inference engine's cycle.

Domain Independence. The ES_Shell is relatively independent of the problem domain of its knowledge base. Any dependencies are a result of the need to interface the knowledge base in a specific way with the environment. For example, the UPDATE module receives incoming bus messages from the aircraft, in the case of the PES knowledge base. These message formats must be predefined, according to the AOS specification. The filters for the incoming data are problem specific as well. The ES_Shell code which must be modified by an application programmer ,however, is limited to the specific portions of UPDATE, dealing with these two functions. The declaration and

initialization of message formats, fact pointers, and data filters are confined to individual packages included within UPDATE to minimize their impact.

Synopsis of Code Developed. The development effort was divided between the design and development of new code and the use or modification of existing code to support the ES_Shell functions. These code components are summarized in Table 2.

Table 2. Developed vs Modified Software

Developed	Used/Modified
Expert System Shell	CLIPS Inference Engine
Knowledge Base	CLIPS Rule Loader/Compiler
AOS Service Interfaces	AOS Services
Update module	
RHS user-defined functions	

The overall system layout showing the relationship between the ES_Shell, the AOS, and the CLIPS/Ada Modules is shown in Figure 26.

Testing with the PES. The sample rule base chosen to exercise the functionality of ES_Shell was the PES. Due to size constraints, only a small subset of the total PES rule base could be used. The TAKEOFFS portion was selected because its twenty rules represent all major knowledge constructs of the PES as well as all major interfaces to the problem environment, i.e. the autonomous air vehicle.

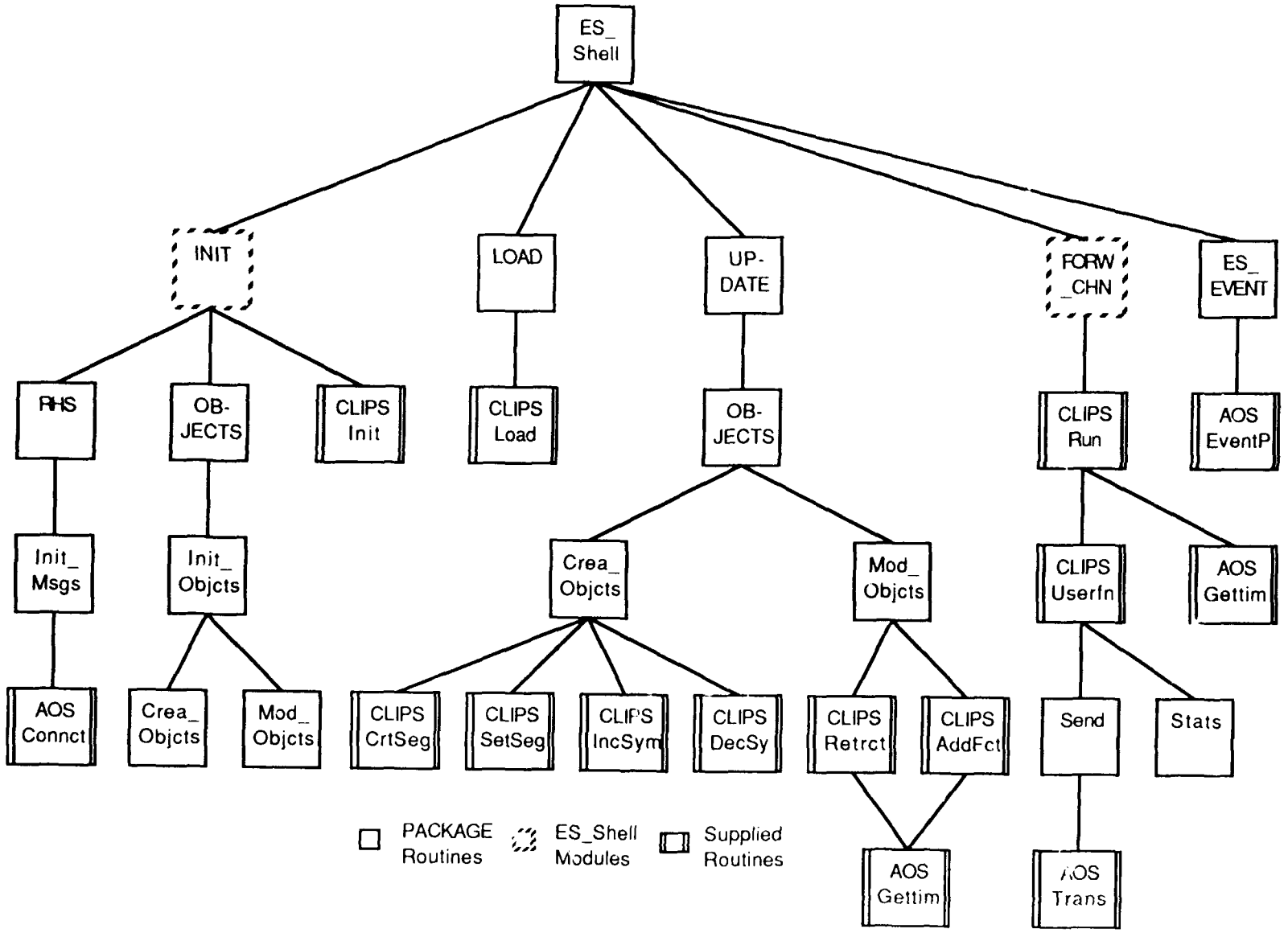

Figure 26. ES Shell System Implementation

A simple aircraft model was used to test the rule base from engine start through takeoff and climb. To achieve this, the inference engine had to reason about the aircraft's current state, and control its performance. Both incoming and outgoing bus messages were needed.

The ES_Shell was tested with the AOS Shell on the VAX. The AOS shell provides the operating system service interface specifications and simple routine stubs to test them with. The update module interfaced to the AOS MESSAGE_IO RECEIVE function SEND used the AOS TRANSMIT services.

The expert system was able to receive data from the aircraft model. This data was filtered by UPDATE prior to assertion in the net. After reasoning about this data, the expert was able to communicate stick and throttle commands to the model to guide it into the air.

The TAKEOFFS expert system successfully demonstrated ES_Shell's ability to interact with the problem environment in a meaningful way, and in an embedded mode. Use of the AOS shell showed that the ES_Shell could use the services of the AOS necessary to accomplish its goal and to act as a callable routine.

The AOS shell contains the interfaces and stubs necessary to test the MESSAGE_IO, TIME and EVENT services used by ES_Shell. The FILE_IO was not implemented in a fashion suitable for testing, so the VAX VMS file services were used instead. To take the ES_Shell further and test it on the VAMPs, the FILE_IO services would have to be used, or the pre-compiled nets linked with the run-time module.

Implications for Future Development. CLIPS contains many features which enhance its use as a development tool. Many error checking and debugging functions, along with an interactive-level inference engine interface, make rapid prototyping rule-based systems

easier. The CLIPS inference engine and fact modification routines allow the introduction of non-static data structures and rule bases. CLIPS also provides its own screen I/O and extended math library packages to enhance portability. Many of these features should be removed, however, for an embedded run-time system, because most of these features come at the expense of code size and execution penalties, as well as data size overhead.

Preparing the ES_Shell itself for VAMP testing should be fairly straightforward. The modular design of the ES_Shell is a nice fit with the VAMP and AOS architectures. Most of the AOS service interfaces have been tested. Bus message interfaces would need to be defined between the expert system and the actual aircraft model of the ITB.

Actual VAMP implementation would depend, however, on the development of a production-quality inference engine. While the CLIPS engine is suitable for development work, it has too many bells and whistles to execute efficiently in the embedded environment, or even fit within the limited memory of the VAMPs.

The inference engine must use an efficient compiled representation of the Rete net. The rule to net expansion for the CLIPS representation of the TAKEOFFS rule base was over 5:1. The storage of tokens in the beta memory of the net requires significant space as well. The VAMPs have only 64 K word data segments and the relatively small (20 rules) TAKEOFFS and few objects (65) take over one third of this space. It may be possible to develop a context switchable net to address the space problem, but then truth maintenance becomes a concern.

From a development perspective, the new production engine should use the same rule syntax or representation as the development systems. This will facilitate maintenance of the knowledge base. This allows full developmental test and debug of the rule base on the development platform, and testing its behavior on the delivery platform.

As previously mentioned, the engine should allow unrestricted access and modification of working memory by both internal and external routines. This will eliminate the arbitrary partitioning of working memory elements required under CLIPS.

If it is assumed that the rule base will be static during execution, no parser or compiler will be needed as well as functions which modify the structure of the Rete net itself. This should payoff with both code size reductions and execution performance increases. Many non-static data structure handling associated with interactive use may be eliminated as the majority of problem attributes are well-defined.

If the knowledge base is thoroughly tested on the development platform, the production engine may eliminate many of the debugging and error handling code commonly found in development systems like CLIPS.

The support of user-defined functions like SEND is essential in order to provide the expert system an interface to the problem environment. The CLIPS method is both convenient and flexible.

Summary. The implementation of the ES_Shell with the CLIPS/Ada and AOS components necessary to support it were presented along with some challenges which had to be overcome during the integration. A successful test of the ES_Shell was accomplished with the PES TAKEOFFS knowledge base on a VAX. The implementation of the ES_Shell exposed many of the weaknesses of the systems whose services it required. In doing so, requirements for future development were made clearer.

Bibliography

Aho, Alfred V., et al. The Design and Analysis of Computer Algorithms. Reading MA: Addison-Wesley Publishing Company, 1974.

Benning, Stephen L., and R. Scott Evans. "Real-Time Operating System for Advanced Avionics Architectures," Proceedings of the IEEE National Aerospace Electronics Conference, NAECON 1988. 138-145. New York: IEEE Press, 1988.

Blair, Jesse L., Chief, Systems Group, Avionics Laboratory, Wright Research and Development Center, Wright-Patterson AFB, OH, 16 October 1990.

Booch, Grady. Software Engineering with Ada (Second Edition). Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc., 1987.

Brassard, Gilles, and Phillipe Bizard. Algorithmics- Theory and Practice. Englewood Cliffs NJ: Prentice-Hall, Inc., 1988.

CLIPS Reference Manual (Version 4.3). Software User's Manual JSC-22948. Artificial Intelligence Section, NASA Lyndon B. Johnson Space Center, Houston, July 1989

Cohen, Norman H. Ada as a Second Language. New York: McGraw-Hill Book Company, 1986.

Common Module Systems. Preliminary Specifications. Digital Processing Systems, Westinghouse Electric Corporation, Baltimore, undated.

Developing Ada Programs on VAX/VMS. Programmer's Guide AA-EF86A-TE. Digital Equipment Corporation, Reading MA, 1985.

Fanning, F. Jesse. Symbolic Processing Configurations for Real-time Embedded Avionic Computer Systems (Addendum). Baseline Memorandum, Project 20031112. System Avionics Division, Avionics Laboratory, Wright Research and Development Center, Wright-Patterson AFB OH, 31 August 1989.

-----. An Introduction to Production Systems and the Rete Algorithm. Unpublished Report to AFIT/EENG 623. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1988.

Forgy, Charles L. and Susan J. Shepard. "Rete: A Fast Match Algorithm", AI Expert, 2: 34-40 (January 1987).

Forgy, Charles L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem", Paper Submitted to Artificial Intelligence, 19:17-37 (Pre-print Release 1982).

-----. On the Efficient Implementation of Production Systems. PhD Dissertation. Carnegie-Mellon University, Pittsburgh, 1979.

Giarrantano, Joseph C. CLIPS User's Guide (Version 4.3). Software User's Manual. Artificial Intelligence Section, NASA Lyndon B. Johnson Space Center, Houston, August 1989.

Guide to VAX Performance and Coverage Analyzer. Software User's Manual AA-EB54E-TE. Digital Equipment Corporation, Maynard MA, 1989.

Gupta, Anoop, and Milind Tambe. "Suitability of Message Passing Computer for Implementing Production Systems." Paper Submitted to the American Association for Artificial Intelligence's National Conference on Artificial Intelligence - 1988 (Pre-print Release), August 1988.

Gupta, Anoop, and Charles L. Forgy. Measurements on Production Systems. Report CMU-CS-83-167. Carnegie-Mellon University, Pittsburgh, 1983.

Hayes-Roth, F. "Rule-Based Systems", Communications of the ACM, 28: 921-932 (September 1987).

Knackstedt Richard P., and Robert H. Brown. Real-Time Artificial Intelligence Support Systems in Ada. Independent Research and Development Report MDC Q0952-12 (Referenced with Written Permission). Space Station Division, Mcdonnell Douglas Space Systems Company, Huntington Beach CA, 15 March 1990. (90029962)

Labhart, Jay T. 1750A Inference Knowledge Base Analysis. Unpublished Report. Merit Technology Incorporated, Plano TX, January 1990.

Labhart, Jay T., et al. MeriTool 2.0 User Manual. User Manual Revision 2.01B, Merit Technology Incorporated, Plano TX, 1989.

Lizza, Captain Carl and Dr. Carl Friedlander. "The Pilot's Associate: A Forum for the Integration of Knowledge Based Systems and Avionics," Proceedings of the IEEE National Aerospace Electronics Conference, NAECON 1988. 1252-1258. New York: IEEE Press, 1988.

Lystaad, Garr S. "The TI Dallas Inference Engine (TIDIE) an Efficient Knowledge Representation for the Piloting Domain," Proceedings of the IEEE National Aerospace Electronics Conference, NAECON 1987. 1348-1351. New York: IEEE Press, 1987.

Martin, John and Dan Neiman. "Rule-based Programming in OPS83", AI Expert, 1: 54-63 (Premier Issue 1986).

McNulty, Christa. "Knowledge Engineering for a Piloting Expert System," Proceedings of the IEEE National Aerospace Electronics Conference, NAECON 1987. 1326-1330. New York: IEEE Press, 1987.

Melebeck, Clovis J. CLIPS/Ada Advanced Programming Guide. Software User's Manual 89-PE-02. Barrios Technology Incorporated, Houston, November 1989.

-----. CLIPS/Ada Architecture Manual. Software User's Manual NAS9-18002. Barrios Technology Incorporated, Houston, November 1989.

Miranker, Daniel P. TREAT: A Better Match Algorithm for AI Production Systems. Report AI TR87-58. Menlo Park CA: American Association of Artificial Intelligence, 1987.

Nayak, Pandurang, et al. "Comparison of the Rete and Treat Production Matchers for Soar (A Summary)," Proceedings of AAAI - 88, Seventh National Conference on Artificial Intelligence. 693-698. Los Altos CA: American Association of Artificial Intelligence, Morgan Kaufman Publishing, distributors, 1988.

Neiman, Dan. "Adding Rete to your OPS5 Toolbox", AI Expert, 2: 42-48 (January 1987).

Nichols, Dave L., and R. Scott Evans. System Specification for the Ada Avionics Real-Time Software (AARTS) Project. Contract Deliverable AARTS-SSS-002. Dayton Engineering Laboratory, TRW, Inc., Dayton OH, 1 September 1987.

Pearl, Judea. Hueristics- Intelligent Search Strategies for Computer Problem Solving. Reading MA: Addison-Wesley Publishing Company, 1984.

Prerau, D.S., "Selection of an Appropriate Domain for an Expert System", AI Magazine, 6: 26-30 (Summer 1985).

Pressman, Roger S. Software Engineering- A Practitioner's Approach. New York: McGraw-Hill Book Company, 1987.

Rich, Elaine, Artificial Intelligence, New York: McGraw-Hill Book Company, 1983.

Shakely, Donald J. Parallel Artificial Intelligence Search Techniques for Real-time Applications. MS Thesis. AFIT/GCS/ENG/87D-24. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

Silberschatz, Abraham, and James L. Peterson. Operating Systems Concepts (Alternate Addition). Reading MA: Addison-Wesley Publishing Company, 1988.

TARTAN Ada VMS V1750A Compilation System (Version 2.0). Programmer' Guide. Tartan Laboratories, Monroeville PA, 1990.

Valdez-Perez, R.E. "Inside an Expert System Shell", AI Expert, 1: 30-42 (October 1986).

van de Goor, A. J. Computer Architecture and Design. Workingham, England: Addison-Wesley Publishing Company, 1989.

VAX Ada Programmer's Run-Time Reference Manual. Programmer's Guide AA-EF88A-TE. Digital Equipment Corporation, Reading MA, 1985.

White, Wesley A. Personal Correspondence. Lead Programmer, CLIPS/Ada, Barrios Technology Inc., Houston, 5 July 1990.
Winston, Patrick H., and B.K.P. Horn. LISP, Reading MA, Addison-Wesley Publishing Company, 1984.

Vita

Captain Franklin Jesse Fanning was born on 4 October 1962 in Laramie, Wyoming. He graduated from Kecoughtan High School in Hampton, Virginia in 1981, going on to attend California State University, Long Beach. After earning the degree of Bachelor of Science in Electrical Engineering in December 1985, he was commissioned a Second Lieutenant in the United States Air Force through the Reserve Officer's Training Corps, and called to active duty in March of the same year.

Then-Lieutenant Fanning was assigned to the Avionics Laboratory, Wright Research and Development Center, Wright-Patterson AFB Ohio, where for two years he programmed real-time display generation systems in the laboratory's Integrated Test Bed facility. For the next three years, he was the team leader of an in-house research project involving integration of expert systems with advanced avionics architectures. He was promoted to the rank of captain in January 1990.

Captain Fanning completed the coursework and research for the degree of Master of Science in Electrical Engineering at the School of Engineering, Air Force Institute of Technology, as a part-time student while assigned to the Avionics Laboratory.

Permanent Address: 1000 Elysian Park Drive
Los Angeles, California
90026

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
AN EVALUATION OF AN Ada IMPLEMENTATION OF THE RETE ALGORITHM FOR EMBEDDED FLIGHT PROCESSORS

5. FUNDING NUMBERS
WU 20031112

6. AUTHOR(S)
F. Jesse Fanning, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/90D-70

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
WRDC/AAAS
System Integration Branch, System Avionics Division
Avionics Laboratory, Wright Research and Development Center
Wright-Patterson AFB OH 45433-6543

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The purpose of this research was to design and develop an expert system shell in Ada, and to evaluate the shell's execution and size performance to determine its suitability for real-time operation on the MIL-STD-1750A embedded flight processor. The expert system shell uses the CLIPS/Ada inference engine, a forward-chaining Ada implementation of Rete. The expert system shell design is presented along with an overview of the target environment- the MIL-STD-1750A VHSIC Avionic Modular Processor (VAMP) running under the Ada Avionics Real-Time Software (AARTS) Operating System. Theoretical and empirical complexity analyses of the inference engine are presented and discussed in view of their impact on VAMP application. The performance of this inference engine was affected by five parameters of the knowledge base: a) the number of objects in working memory, b) the structural complexity of the objects and rules, c) the number of rules which share object match patterns, d) the number of match patterns per rule, and e) the number of objects bound to a match pattern using negative logic. The inference engine's execution response time was found suitable for real-time operation on the VAMP; however, its memory requirement was not.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Expert Systems, Ada, Embedded Flight Processors, AARTS, VAMP	114
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL